Formal Object Oriented Development of Software Systems using LOTOS.

Article · December 1997

Source: CiteSeer

1 author:

J. Paul Gibson
Institut Mines-Télécom
92 PUBLICATIONS **570** CITATIONS

SEE PROFILE

Formal Object Oriented Development of Software Systems using LOTOS.

J. Paul. Gibson
Department of Computing Science,
University of Stirling,
Stirling FK9 4LA.

A thesis submitted in partial fulfilment of the
requirements for the Degree of Doctor of Philosophy
in Computer Science at Stirling University

July 1993

Abstract

Formal methods are necessary in achieving *correct* software: that is, software that can be proven to fulfil its requirements. Formal specifications are unambiguous and analysable. Building a formal model improves understanding. The modelling of nondeterminism, and its subsequent removal in formal steps, allows design and implementation decisions to be made when most suitable. Formal models are amenable to mathematical manipulation and reasoning, and facilitate rigorous testing procedures. However, formal methods are not widely used in software development. In most cases, this is because they are not suitably supported with development tools. Further, many software developers do not recognise the need for rigour.

Object oriented techniques are successful in the production of large, complex software systems. The methods are based on simple mathematical models of abstraction and classification. Further, the object oriented approach offers a conceptual consistency across all stages of software development. However, the inherent flexibility of object oriented approaches can lead to an incremental and interactive style of development, a consequence of which may be insufficient rigour. This lack of rigour is exacerbated by the inconsistent and informal semantics for object oriented concepts at all stages of development.

Formal and object oriented methods are complementary in software development: object oriented methods can be used to manage the construction of formal models and formality can add rigour to object oriented software development. This thesis shows how formal object oriented development can proceed from analysis and requirements capture to design and implementation.

A formal object oriented analysis language is defined in terms of a state transition system semantics. This language is said to be *customer-oriented*: a number of graphical views of object oriented relations in the formal analysis models are presented, and the specifications produced say *what* is required rather than *how* the requirements are to be met. A translation to ACT ONE provides an executable model for customer validation. This translation is founded on a precise statement of the relationship between classes and types (and subclassing and subtypes). The structure of the resulting ACT ONE requirements model corresponds to the structure of the problem domain, as communicated by the customer.

The step from analyis to design requires an extension to the requirements model to incorporate semantics for object communication. A process algebra provides a suitable formal model for the specification of communication properties. LOTOS, which combines ACT ONE and a process algebra in one coherent semantic model, provides a means of constructing object oriented design semantics. Design is defined as the process of transforming a *customer-oriented* requirements model to an *implementation-oriented* design, whilst maintaining *correctness*. Correctness preserving transformations (CPTs) are defined for: transferring requirements structure to design structure, manipulating design structure and changing internal communication models.

Design must be targetted towards a particular implementation environment. The thesis examines a number of different environments for the implementation of object oriented LOTOS designs. It illustrates the importance of understanding programming language semantics. We show how Eiffel can be used to implement formal object oriented designs.

A case study which evaluates the formal object oriented models and methods, developed in this thesis, is reported. This identifies re-use at all stages of software development and emphasises the role of structure: it improves understanding and communication, and makes validation and verification *easier* and *better*.

The thesis shows that formal object oriented technology is ready for transfer to industry. These methods should be exploited sooner rather than later: object oriented development can incorporate formal methods without significant cost, and formal methods can utilise the object oriented paradigm to manage complexity. The thesis provides a rationale for formal object oriented development and a set of conceptual tools which makes the development of software systems a true *engineering* discipline.

Declaration

I hereby declare that this thesis has been composed by myself, that the work reported has not been presented for any university degree before, and that the ideas I do not attribute to others are due to myself.

Paul Gibson
July 1993

Acknowledgements

The completion of this thesis was dependent on many different people, my thanks goes to everyone who encouraged me in this work, even if I forget to mention them by name.

My supervisor, Ken Turner, must be acknowledged for the time and effort which went into my supervision. Ashley McClenaghan must also be thanked for his friendship and support whilst sharing a room with me for almost four years. The computing science department at Stirling University is thanked as a whole, but, in particular, Sam, Graham and Catherine (for their technical support), and Moira and Jane (for their secretarial expertise), and Bob Clark (for ably fulfilling the role of a second supervisor) must be acknowledged.

British Telecom (Research and Development) are acknowledged for their financial contributions, together with the Department of Education for Northern Ireland. BT also offered support in the form of technical advice and personal encouragement: Elspeth Cusack, Steve Rudkin, Jim Lynch, Steve Colwill, Jeremy Wilson, Rob Booth, David Freer and many others are thanked in this respect.

On a personal note, my family are thanked for their support of an *eternal student*: my mother, father, David and Andrew have given me more help than they could know. My friends: Geoff (who never had a *cross word*), Flash, Andy, Steve, Dave, Gary, Richard, Ana, Peter, Dominique and Bazza have helped with their sense of humour and understanding.

Finally, I would like to thank my girlfriend, Jane, for putting up with me whilst completing this work: she may not understand what it means, but she, more than anyone, understands what it means to me!

I finish with a sentiment (a lesson from my parents which I have only just fully learned):

> When you value something by how much it has cost then you don't really value it at all.

Contents

List of Figures

Chapter 1

Introduction

Formal languages, based on mathematical models, are essential in achieving *correct* software. *Correctness*, being a mathematical proof that software fulfils requirements, depends on: a formal requirements model, a formal implementation model and a means of relating the two. These requirements appear straightforward until the complex nature of software is considered.

Software development has reached the point where the complexity of the systems being modelled cannot be handled without a thorough understanding of underlying fundamental principles. Such understanding forms the basis of scientific theory as a rationale for software development techniques which are successful in practice. This scientific theory, as expressed in rigorous mathematical formalisms, must be transferred to the software development environment. Only then can the development of software systems be truly called *software engineering*: the application of techniques, based on mathematical theory, towards the construction of abstract machines as a means of solving well defined problems.

Object oriented methods encompass a set of techniques which have been, and will continue to be, applied in the successful production of complex software systems. The methods are based on the simple mathematical models of abstraction and classification. Further, the conceptual consistency offered by the object oriented paradigm across the software development process, together with an emphasis on re-use, promotes very fast code production cycles. However, the inherent flexibility in object oriented development environments often leads to an interactive and incremental style of development, a consequence of which may be insufficient rigour in software production. This lack of rigour is re-inforced by the informal set of object oriented concepts and terminology which, although applied in all stages of development and based on the same underlying principles, mean different things to different people.

Object oriented methods can be used to aid the construction of formal models. Formality can help to improve object oriented software development techniques. Object oriented and formal methods are complementary: their integration has the potential for producing a software development environment which incorporates the benefits of both. Fundamental to the successful integration of mathematical rigour and object oriented methodology is a formal model of object oriented concepts which is consistent throughout the whole software development process. Such a model forms the basis

of the work put forward by this thesis.

1.1 Scope: Software Engineering

Software engineering, formal methods and the object oriented paradigm together form an enormous body of work, much of which is beyond the scope of this thesis. The software engineering boundaries of the work are clearly defined in figure 1.1. The analysis, requirements capture, design and implementation models and methods put forward by this thesis are collectively named FOOD: *Formal Object Oriented Development.*

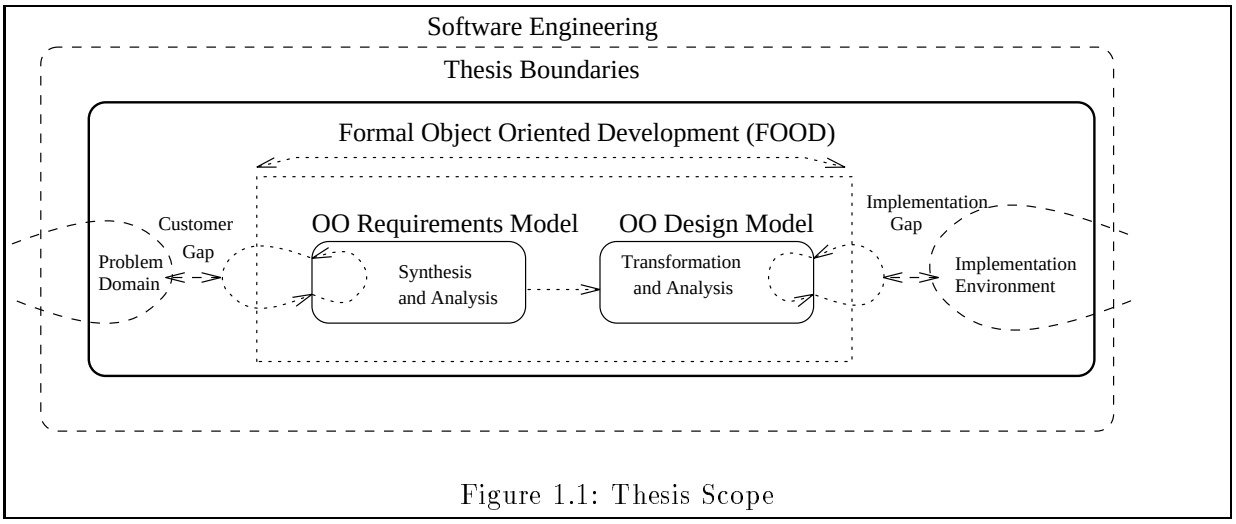


Figure 1.1: Thesis Scope

FOOD is principally concerned with maintaining correctness between the initial *customer oriented* requirements model and the final *implementation oriented* design. The formal boundaries break down at either end of the software development process because, in general, target implementation languages are not formally defined and customer understanding of their requirements is not complete. An object oriented approach helps to bridge these two gaps because of the conceptual consistency.

The thesis makes a clear distinction between models and methods: models provide only a framework upon which software can be constructed, whilst methods define ways in which models can be synthesised and analysed to aid all stages of development. Although FOOD puts forward analysis and requirements capture, design and implementation strategies, these do not define an industrial strength software development method, which is beyond the scope of this thesis.

1.2 Context: Structured Development, Formal and Object Oriented Methods

The context of the work in this thesis is represented in the diagram in figure 1.2. The three main areas of interest, within software engineering, are structured software development methods, formal methods and object oriented methods. Within these areas we are most interested in formal object

oriented development (FOOD) and the role of the specification language LOTOS. These five areas are overviewed in sections 1.2.1 to 1.2.5, below[1].

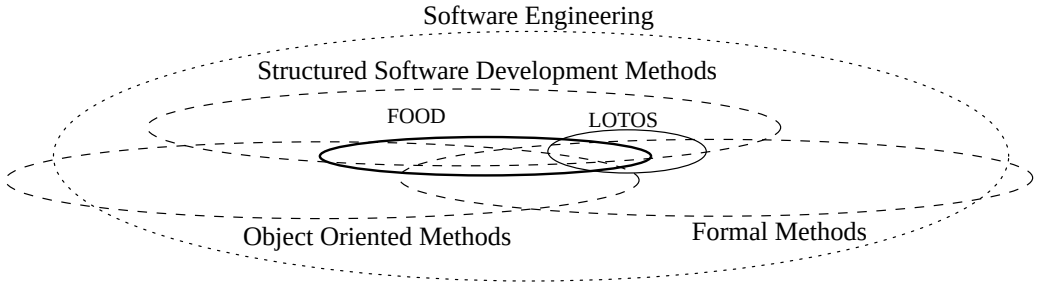


Figure 1.2: An Overview of the Problem Domain Structure

1.2.1 Structured Software Development Methods

Structured software development techniques have emerged in response to the growing complexity of the systems being modelled. Constraints on the complexity of systems which can be handled have gradually moved from hardware to software. Consequently, programming languages have had to evolve to cope with much larger problems. In the early 1970s, the need for both methodological and formal approaches to programming were identified in a number of different texts, the best known of which are [40, 44, 127, 45, 105, 116]. High-level programming languages encouraged *structured programming*: the development of programs composed of (sub)programs composed of etc. ..., combined with the ability to share complex data structures between programs (Modula-2 [128] is a good example of a language which encourages structured programming [18]). Structured programming leads to a natural functional decomposition of a problem. This is reflected in the large number of software design methods which place emphasis on functional structure. Good examples of these methods are found in [27, 90, 94, 36, 41, 51, 80]. A good overview of the differences and similarities between these methods is found in [17].

The functionally based structured software development methods placed emphasis on data flow modelling (in various forms), which shows the transformation of data as it flows through a system. The transformations are carried out by functions and design involves the repetitive division of functions into (sub)functions until the lowest-level components are simple enough to implement directly. This simple view of the functional divide-and-conquer approach does not do justice to the complexity inherent in many of the structured methods, but it is the underlying strategy in each of these techniques.

There are many reasons why functional approaches are in such wide use:

- Programmers think in terms of functions and so find the transition to structured analysis and design methods easy to make.

[1] These sections give an overview of, and references to, some of the most important work related to the thesis. More comprehensive references are given in the appropriate parts of the main body of the thesis. It was considered counterproductive to give references to all work at this introductory stage.

- Functionally based methods were the first widely available and well documented techniques.
- The techniques offer good project management support, which plays a vital role in software development [49, 66].
- The methods are founded on graphical models, which appeal to the users.

However, as the rest of this thesis argues, approaches based on functional decomposition (data flow) are, in general, *inferior* to methods based on data structure. Further, the inherent informality in the most popular structured methods is not easily overcome.

1.2.2 Formal Methods

Many software engineers do not acknowledge the value of formality. This thesis is founded on the belief that formal methods are *just about* ready for transfer to general software development in industry. This belief is supported by a major study of the current state-of-the-art in formal methods [32] which concludes by stating:

> " ...formal methods, while still immature in certain important respects, are beginning to be used seriously and successfully by industry to design and develop computer systems ..."

This reinforces the statements made in [50, 71, 5] concerning the ever increasing importance of formality in the software development process.

There are a wide and varied range of definitions of *formal method* which can be found in the majority of texts concerned with mathematical rigour in computer science. (A wide range of *formal methods* are considered in [46, 115, 3, 96, 72, 19].) For the purposes of this thesis we propose the following definition:

> *A formal method is any technique concerned with the construction and/or analysis of mathematical models which aid the development of computer systems.*

Formal methods are fundamentally concerned with *correctness*: the property that an abstract model fulfils a set of well defined requirements. This notion is addressed in a number of different texts and with respect to a number of different models, see [9, 10, 8, 15, 43, 34] for example.

A major problem when using formal methods in software engineering is that much of the recent research places emphasis on analysis rather than synthesis. The means of constructing complex formal models is often overlooked in favour of techniques for analysing models. In this thesis, emphasis is placed on the construction of formal models. Formal method engineers need to learn techniques for building very large, complex systems. Such techniques have been followed, with various degrees of success, by programmers. In particular, object oriented programmers have evolved techniques which have been successfully transferred to the analysis and design phases of software engineering. Where better then to look for aid in the construction of large formal models?

1.2.3 The Object Oriented Paradigm

The object oriented paradigm arose out of the realisation that functional decomposition is not the only means of structuring code: an alternative is to construct a system based on the structure of the data[2]. Emphasis on data structure led to the encapsulation of functional behaviour within data entities: *objects*[3].

Object oriented concepts were conceived in Simula [91], went through infancy in Smalltalk [58, 57] and could be said to be leaving adolescence, and approaching maturity, in the form of many different languages (for example: Objective C [31], C++ [106], LOOPS [7], Flavours [21, 88], CLOS [42, 73], Eiffel [84] and Common Objects [103]) and methods (for example: those of Rumbaugh [101], Coad and Yourdon [25, 26], Cox [31], Meyer [84] and Booch [13, 12]). [89] provides a good review of object oriented languages and methods with respect to object oriented analysis and design. Unfortunately, none of the well accepted methods provides a formal framework upon which the work in this thesis could be based.

1.2.4 Formal Methods and Object Orientation

There has been much interest in combining formal and object oriented methods. The research falls into two main categories:

- **i) Using Object Oriented Techniques To Construct Formal Models**
  The success of object oriented techniques in software development has led to much interest in using the same techniques for building formal models. Much of this work centres on the definition of object oriented constructs, or the interpretation of object oriented concepts, in an existing formal language. Good examples of the type of work which has been done can be found in [24, 6, 33, 75, 100, 118, 56, 97, 81, 77]. This work has led to recognition of the inconsistent use of object oriented terminology, highlighting the need for a concensus of opinion. Further, much of the work shows the difficulties inherent in modelling object oriented behaviour in a semantic framework which was not designed for such a purpose.
- **ii) The Development of Object Oriented Semantics**
  The lack of agreement on the meaning of object oriented constructs, reinforced by the informal semantics of most object oriented programming languages, has led many people to produce formal object oriented semantics, for example see [14, 130, 95, 123, 47]. The thesis by Wolczko [129] provides a more complete view of the technical issues, whilst Wegner [124] and America [1] examine the philosophical aspects. Much of this research has had a positive influence on the work in this thesis. However, the semantics examined were not deemed suitable for use in this work because they did not fulfil our three main requirements (apart from formality):

[2]Of course, there are programming languages which do not place emphasis on functional or data structure, but we do not consider them in any detail as part of this work.

[3]Two well known data-based sofware development methods which are generally accepted as not being object oriented are the quite similar approaches put forward by Jackson [69] and Orr [93]. These approaches are closely related to the object oriented paradigm in the initial analysis stages, but digress from the standard object oriented view as they approach implementation.

- 1) We require a semantics which agrees with our intuitive understanding of object oriented concepts, as recorded in chapter 2.
- 2) We require a semantics which, with a suitable syntax, is accessible to the customers for use during analysis[4].
- 3) We require a semantics which can be used during all stages of object oriented development, in other words we need a wide-spectrum language.

Given these very specific requirements, it was necessary to define a new semantic framework.

The work in this thesis has a foot in both these areas, which share common ground. We develop an object oriented semantics (based upon a constructive, easy to understand, state transition system model). The abstract data type (ADT) part of LOTOS (*Language Of Temporal Ordering Specifications*), see [15, 112, 113, 68, 117], is used to implement the requirements models which are defined using this semantics. Then, *full LOTOS* (LOTOS with the ADT and process algebra parts) is used to model the requirements in a more concrete framework. In the thesis, LOTOS specifications are constructed using object oriented techniques and LOTOS is used to define a high-level object oriented semantics: these views are complementary.

In this thesis, emphasis is placed on semantic continuity. The object oriented semantics, based on the labelled state transition model, are present throughout development. LOTOS is used only to make the object oriented models more concrete, as development approaches implementation.

1.2.5 LOTOS

LOTOS is chosen as the object oriented requirements capture and design language because:

- Its natural division into ADT part (based on ACT ONE[5] [48]) and process algebra part (similar to CSP [65] and CCS [87]) suits our need for semantic continuity from analysis to design: the ACT ONE requirements model can be incorporated within the full LOTOS design model.
- LOTOS has already been the subject of research with regard to its suitability for modelling object oriented systems and incorporating object oriented principles: for example, see [6, 118, 100, 81, 75, 33, 24].
- LOTOS is a wide spectrum language, which is suitable for specifying systems at various levels of abstraction. Consequently, it can be used at both ends of the design process.
- There is wide support, often in the form of tools, for the static analysis and dynamic execution of LOTOS specifications: for example, see [117, 61, 10, 92].

Although ACT ONE and LOTOS are prominent throughout FOOD, the main work in this thesis revolves around the principles rather than the languages used to implement the principles. Any abstract data typing language has the potential to implement the requirements models. Similarly,

[4] We argue, in later chapters, that this is possible only if the requirements models being used are constructive.

[5] In the remainder of this thesis the ADT part of LOTOS is referred to as ACT ONE, even though this identity is not quite precise.

any process algebra can be used to specify the internal and external communication models which are introduced in design. The problem of going from analysis to design (abstract types to more concrete processes) is made easier in LOTOS because of the way in which the two different parts of the language are integrated.

1.3 Formulation of an *Ideal* Development Environment

In an ideal software development environment the following requirements must be met:

- **Consistency**
  There must be a consistency in the models of conceptualisation used throughout development. Further, there must be a coherent approach which can be applied consistently in different problem domains.
- **Correctness**
  It must be possible to guarantee that the final *implementation oriented* design is *correct* with respect to the initial *customer oriented* requirements model.
- **Re-Use**
  It is vital that re-use is prominent at all stages of software development, see [98, 52]. When constructing any engineered artifact it is desirable to use *materials* whose behaviour is well understood. In software engineering these *materials* are available in many different forms: low-level language constructs, medium-level predefined code components and high-level development methods.
- **Opportunism**
  The software development method should not be too prescriptive. A developer should be able to do what seems best at any stage of development within well defined bounds, and the method should support this. The lack of opportunistic flexibility often constrains software development, see [119, 76, 62, 79]. Uncontrolled opportunism is not desirable, but there is no reason why software developers cannot be encouraged, by the method being employed, to both *craft* and *engineer* software.
- **Customer Awareness**
  The analysis and requirements capture phases of software development should be *customer oriented*: it is generally agreed that customer communication is the most important aspect of analysis [67, 99, 114]. The successful synthesis of a requirements model is dependent on being able to construct a system as the customer views the problem [54]: requirements validation is not possible if the models cannot be communicated to the customer.
- **Implementation Awareness**
  It is important that the design process can be targetted towards a wide range of implementation environments. In particular, the implementers must be able to make a correspondence between constructs in the implementation domain with constructs in the final design.

In industry, there are other requirements of a software development environment which, although important, are not listed above. We believe that meeting the six requirements, above, provides a framework upon which all other requirements can be met.

1.4 Formal Object Oriented Development (FOOD): Prototyping An *Ideal*

1.4.1 Fulfilling The *Ideal* Requirements: An Overview

Combining formal and object oriented methods provides a framework upon which the *ideal* development requirements can be met:

- **Object Oriented ⇒ Consistency**
  The object oriented paradigm provides a conceptual consistency throughout the development process. It is proposed that one abstract model of object oriented concepts is maintained throughout analysis, design and implementation. Certainly, the concepts are then realised in more concrete terms as the models approach implementation, but the underlying abstract semantics are constant. Experience suggests that object oriented techniques are applicable in most problem domains [110], and consequently their use can be consistent from project to project[6].
- **FOOD: The Road To Correctness**
  There are three important aspects to the development of correct software:
  - i) Validating Customer Requirements
    A formal requirements model means that the customer does not have to deal with ambiguity, inconsistency or incompleteness. However, a formal model does not guarantee correct customer comprehension: the model can be wrongly interpreted[7]. An object oriented model is less likely to be misunderstood since it records the requirements in an intuitive way.
  - ii) Correctness Preserving Design
    Design is required to take an abstract, *customer oriented* model and produce a more concrete, *implementation oriented* model whilst preserving *correctness*. This can only be done when the two models are formally specified. The present state-of-the-art in formal methods cannot cope with the task of verifying any given formal design against any given formal requirements model. Consequently, this thesis advocates a transformational approach to

[6] Whilst this thesis agrees with the general applicability of object oriented techniques, we believe that other approaches are superior in particular problem domains. It is beyond the scope of this thesis to identify these areas.

[7] A further problem occurs when the customer misunderstands their requirements, rather than misunderstanding the requirements model. Such complexities are beyond the scope of this work, but we do believe that the process of constructing a formal object oriented requirements model reduces the risk of this occurring because of the resulting improvement in mutual understanding between customer and analyst.

design: design is defined as a sequence of transformation steps which reflect design decisions for moving from the abstract to the concrete. Each transformation must preserve correctness, i.e. the designers must be prepared to show that the latest design satisifies the requirements imposed by the previous stage. Correctness preserving transformations (CPTs) are a means of automatically guaranteeing *correct* design steps. It is important to note that there must still be a gap between the final requirements model and initial design (from *what* to *how*). However, a formal object oriented approach means that this gap can be bridged in a rigorous fashion.

- iii) Testing the Implementation
  Presently, most implementation languages are not formally defined. Consequently, one cannot, in general, prove the correctness of an implementation with respect to a formal design. The formal object oriented approach does, however, make the testing of implementations much more rigorous: the code can be verified against unambiguous requirements and the object oriented conceptual consistency means that there is a structural correspondence between design and implementation. This promotes a compositional approach to testing. Further, the emphasis on re-use means that much of the implemented system has already been tested elsewhere.

- **Object Oriented $\Rightarrow$ Re-Use**
  The object oriented paradigm gives rise to two different types of re-use: composition and classification. Classes and class hierarchies provide re-usable components at all stages of development and at different levels of abstraction, see [83, 70]. Care, however, must be taken when inheritance is used as a code re-use facility [108].
- **Object Oriented Methods Support Opportunism**
  There is an inherent flexibility in object oriented development:

  - Object oriented development is both bottom-up and top-down, supporting composition and decomposition at all levels of abstraction. Developers, therefore, are not restricted to a simple repetitive analysis-synthesis-test sequence.
  - The scalability of object oriented methods and consistency between all stages of development means that different parts of a system may be at different levels of abstraction without undue complication. Consequently, developers can easily move between system parts in a flexible fashion.

- **Customer Awareness**
  The need for *customer oriented* models has already been emphasised in this chapter. Object oriented models certainly reflect the way in which customers comprehend their problem, but not all object oriented models are accessible to customers, particularly those with formal semantics. It is important that these models are presented in the form of *customer oriented* notations: in practice, this means providing graphical representations of the formal models [53]. The object oriented paradigm has a number of similar, but inconsistent, graphical representations

associated with particular development methods. These types of diagram can be provided in a formal framework without diminishing their ability to improve customer communication.

- **Implementation Awareness**
  Formal methods can provide designers with enough power to target the designs towards a wide range of implementation languages and environments. Further, object oriented designs can be implemented in a wide range of programming languages, not just those which are object oriented.

This section has argued that a formal object oriented approach has the potential to provide an *ideal* software engineering environment. Figure 1.3 contains more details with respect to the components (models and methods) which are used, in this thesis, to construct a framework for the development of such an *ideal*. This prototype construction is certainly not the only (or *best*) way of providing such a framework, but it does show the feasibility of developing an industrial strength formal object oriented software development environment and illustrates the advantages in doing so.

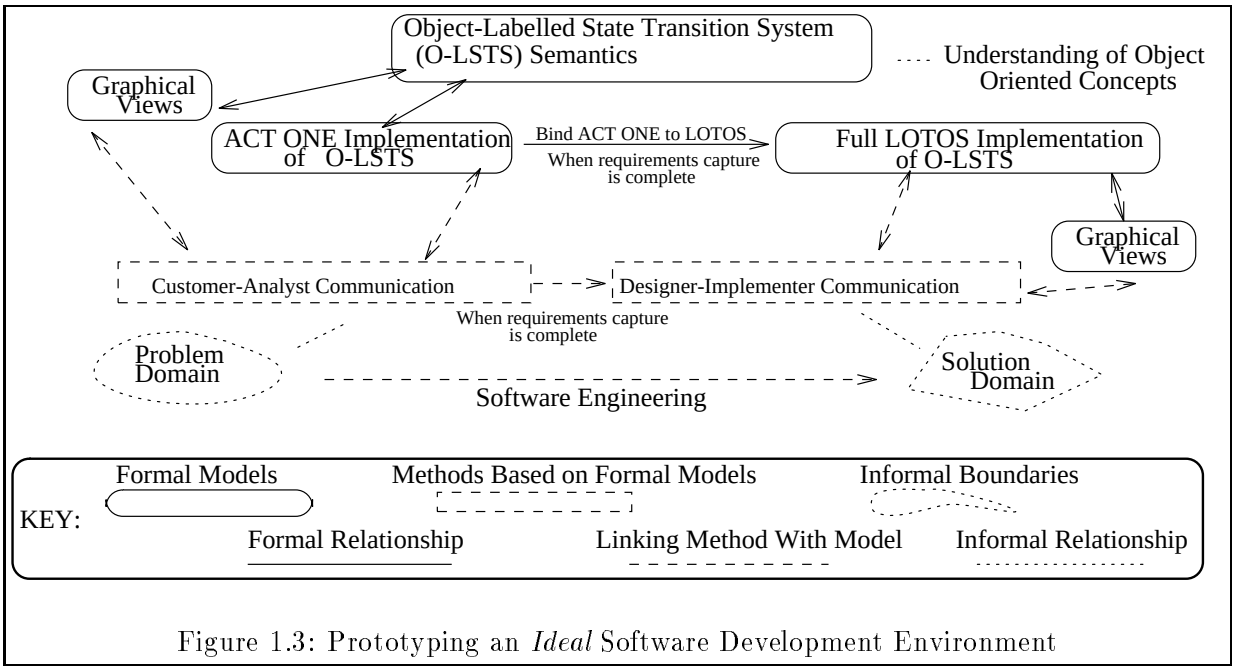


Figure 1.3: Prototyping an *Ideal* Software Development Environment

1.4.2 A Step-by-step Construction of the FOOD Environment

The main body of this thesis constructs a prototype of an *ideal* software development environment, based on the framework outlined in figure 1.3. The work naturally progresses from analysis to design and onto implementation. An overview of the structure and contents of each of the remaining chapters of this thesis is given below.

Chapter 2. Analysis: Modelling Problem Understanding

This chapter introduces formal object oriented analysis and the object oriented paradigm. An overview of software analysis methods places emphasis on problem domain understanding, customer

communication and requirements notation. A list of features which should be evident in a *good* analysis method are then identified. The object oriented paradigm is investigated as a means of providing a framework upon which all these features can be offered. The need for formal analysis models is introduced, and abstract data types (ADTs) put forward as a natural means of modelling classes of objects during analysis. This leads to an initial investigation of the relationship between classes and types. The chapter concludes with a preview of the formal object oriented analysis models which are developed in chapter 3.

Chapter 3. An Object Oriented Semantic Framework

This chapter formalises the understanding of object oriented concepts arising from the investigation in chapter 2. An abstract object oriented semantics is defined as a particular type of state transition system. From this simple basis the formal definitions of class, object, attributes, subclassing, composition, configuration and interaction are derived. The semantics are then syntactically sugared, placing emphasis on the object oriented concepts, to provide a formal object oriented analysis language which is accessible to customers and analysts alike. Graphical views of the object oriented properties, implicit in models defined using this language, are developed. An executable model of the analysis language semantics is provided by a translation to ACT ONE.

Chapter 4. Formal Object Oriented Analysis: The Practical Issues

This chapter examines practical issues which arise when analysing and synthesising the formal object oriented requirements models. Emphasis is placed on the construction of classification and composition hierarchies, with particular regard given to the difference between subclassing and delegation. Communication and interaction are considered at an abstract level: *what* rather than *how*. Other analysis issues, namely nondeterminism, exceptions and implementation freedom are also examined. A method, in the form of a high-level algorithm, is given for the development and validation of object oriented requirements models. Finally, the step from requirements to design is previewed.

Chapter 5. Formal Object Oriented Design (Using LOTOS)

This chapter begins by introducing design and highlighting the importance of learning from different design areas. Software design is then considered, with a focus on object oriented methods. This is followed by an overview of LOTOS and its suitablity for modelling object oriented systems. Then, the means of going from an ACT ONE requirements model to a full LOTOS design is examined. The importance of maintaining correctness across the design process is stated. With this in mind, the notion of correctness preserving transformation (CPT) is introduced. Finally, a small set of CPTs are defined to correspond to decisions that are commonly made during object oriented design. Design is then defined as the process of transforming a *customer oriented* requirements model to an *implementation oriented* model, using CPTs wherever possible. The target implementation model is shown to have a great influence on the design decisions.

Chapter 6. Object Oriented Program Derivation

The implementation of high-level object oriented LOTOS designs is considered. Many different target implementation environments are examined, not just those which are object oriented, with respect to their suitability for use in the final implementation stage of FOOD. It is argued that object oriented programming languages provide the easiest targets at which object oriented designers can aim. Eiffel [84] is chosen, from a wide range of object oriented programing languages, to illustrate the implementation process. The importance of having a thorough understanding of programming language semantics is stated. The informal Eiffel semantics are reviewed and a methodological, but informal, technique for generating Eiffel code from object oriented LOTOS designs is proposed. Finally, the future generation of concurrent implementations is identified as an area which is well supported by FOOD.

Chapter 7. Formal Object Oriented Development: A Case Study

This chapter puts all the theory from chapters 2 to 6 into practice. A small banking network system is developed using the analysis, requirements capture, design and implementation models and methods proposed by this thesis. The case study does not utilise, or consequently test, all aspects of FOOD, but it does show that the method has the potential for use in real software development projects. To conclude the chapter, a list of the more important lessons arising from the case study is given. The most important lesson is that much more work remains to be done.

Chapter 8. Conclusions

This chapter concludes the thesis by reviewing the initial objectives, showing how the thesis meets these objectives and identifying areas of further work arising out of the thesis.

1.5 Contributions of the Thesis

The main contributions of the thesis are: the philosophy and reasoning behind a formal object oriented development strategy, the object oriented mathematical models resulting from this reasoning, the software engineering methods which utilise the models in a consistent and coherent fashion (collectively called FOOD) and the preliminary case study which shows the effectiveness of these methods.

1.5.1 FOOD: The Philosophy

The philosophy upon which FOOD was developed is a major contribution of the thesis:

> Formal and object oriented methods are complementary. *Correctness* is the most important property of software. Formality is the only means of guaranteeing *correctness*. The complexity of constructing software models (at all levels of abstraction) can be managed using object oriented techniques. Formal object oriented techniques help to bridge the

informal gaps at either end of software development. Formal object oriented development is *software engineering* in a pure form.

1.5.2 FOOD: The Models

The need for precision when defining object oriented concepts leads to the development of a number of formal models. An abstract object oriented semantics is developed to form the basis of our understanding of all object oriented models (from analysis to implementation). *Customer oriented* analysis and requirements models are defined, together with a formal means of stepping from requirements models to initial designs. Correctness preserving transformations are defined to reflect object oriented design decisions. The informal step from final design to an implementation model is strengthened by a rigorous investigation of the semantics of Eiffel (the chosen implementation language). The way in which LOTOS is used throughout FOOD is both original and effective.

1.5.3 FOOD: The Method

The emphasis during software development using FOOD is on rigour, re-use and opportunism. A skeleton method is provided for the requirements capture, design and implementation stages. This method is not yet strong enough for industrial use[8], but there is potential for either integrating FOOD with other more commercial techniques or extending FOOD with less technical but more practical constructs. The method has been used in a trial case study to illustrate its effectiveness.

Contribution Summary

As a whole, the thesis offers a clear and concise statement of the problems inherent in software development, together with proposals for solutions to these problems which are based on the integration of formal and object oriented methods. A framework (FOOD) is developed for the implementation of these solutions.

[8] We believe that a method should evolve from experience of using the models.

Chapter 2

Analysis: Modelling Problem Understanding

The work in this chapter is structured as follows:

- **Section 2.1: Introducing Formal Object Oriented Analysis (FOOA)**
  This section provides a brief review of object orientation and formalisation with respect to the limitations of current analysis methods. It motivates the development of a formal object oriented analysis method.
- **Section 2.2: Analysis: An Overview**
  This section begins with a more complete overview of different analysis methods and models. It proceeds to define a list of criteria for judging analysis techniques. Based on these criteria, we propose formal object oriented analysis as a natural successor to more traditional approaches.
- **Section 2.3: Object Oriented Analysis: An Informal Approach**
  This section examines object oriented analysis from an informal point of view. A number of simple example systems are analysed and these illustrate the types of properties that a customer is likely to want to express in an object oriented framework of understanding. This section concludes by identifying five analysis models which combine in a coherent fashion to give a complete view of object oriented requirements.
- **Section 2.4: Formal Object Oriented Analysis Using Abstract Data Types**
  This section argues that there are advantages in using abstract data types (ADTs) during requirements capture and analysis. It reviews the informal links that already exist between ADTs and the early stages of object oriented development.
- **Section 2.5: Classes and Types**
  This section examines the more practical issues that arise when comparing class with type. In particular, it distinguishes between the different roles that each concept plays within the frameworks in which they are found.
- **Section 2.6: A Formal Object Oriented Requirements Model in ACT ONE: A Preview**

This section previews the work in chapter 3 by reviewing the way in which we propose to model object oriented requirements using ACT ONE. ACT ONE is not used as our object oriented analysis language: it serves only as a semantic model onto which object oriented requirements are mapped.

2.1 Introducing Formal Object Oriented Analysis (FOOA)

Requirements capture and analysis (RCA), within software development, is the first step in the long, often arduous, process of satisfying the needs of the customer. In short, it is the process of identifying and recording what is required. Unfortunately, the RCA process must fulfil two very different roles:

- The customer must be convinced that the requirements are completely understood and recorded.
- The designers must be able to use the requirements to produce a structure around which an implementation can be developed and tested.

The requirements act as an interface between problem domain 'experts', with potentially very little comprehension of computers, and solution domain professionals, who understand computer systems, languages, models and techniques but may have little knowledge of the problem environment. This dual role makes RCA a not insignificant problem. However, there is much hope in the knowledge that many of the same principles of structuring, organisation and method are found in both domains: the common theme is complexity management.

2.1.1 Introducing Traditional (Functional) Approaches

Traditional analysis methods (see [67, 99, 51, 36], for example) define ways in which to control complexity using a functionally oriented divide-and-conquer strategy. These methods, in general, have two major deficiencies:

- **They lack formality.**
  Being informal, they are open to interpretation and inhibit rigorous means of validation and verification. Lack of formality makes the notion of contractual software less appealing. Code re-use, which is dependent on the existence of libraries of well-defined components, is possible only through a formal statement of behaviour and a means of classifying behaviour to facilitate access to appropriate components.
- **The analysis does not lead to one consistent model.**
  Traditionally, the modelling of requirements results in an unnatural division between process and data. This separation of concerns can lead to two models which, at best, are difficult to relate, and, at worst, are contradictory.

2.1.2 Object Orientation

Object oriented techniques and concepts have been shown to be applicable in the analysis phase of development (see [25], for example). This should not be surprising since object oriented programming

is often said to be 'real world modelling' [31] which, in general, is what analysts are doing. The idea of applying object oriented methods, which initially grew up in the programming domain, to design and analysis corresponds to the way in which structured approaches, in the 1970's, gradually infiltrated each stage of software development.

The object oriented philsophy does not throw away all the previous work on structured analysis; it re-uses many of the ideas and combines them in a consistent and coherent fashion. We argue that the application of object oriented methods does not make requirements capture easy, but it does make it easier. Object oriented techniques can be applied to different systems with much greater confidence in the underlying principles. In this way, the method becomes second nature and understanding of the systems being analysed is given prominence. With traditional analysis techniques, the balancing of the process and data parts of the problem inhibits understanding. Consequently, the structure of the problem domain is often compromised. The object oriented approach promotes the maintenance of problem domain structure throughout the whole development process. It is the conceptual integrity of the object oriented paradigm which provides the essential bridge between customer requirements and program design.

2.1.3 Formalisation

In light of the previous section, one could be forgiven for believing that object orientation does everything you could ever want (and more). However good object oriented methods are at modelling real world requirements, they do not provide a formal framework for mathematical reasoning and manipulation. Like traditional approaches, the diagrams central to object oriented methods are not formally defined. The strength of these diagrams is that the customer finds them easy to understand. This is also the root of their weakness: by ensuring 'lay-person' readability, the informal approaches lose much of their potential for reasoning and manipulation.

A formal model of requirements is unambiguous — there is only one correct way for designers to interpret the model. Although the model must still be mapped onto the real world, this mapping is in essence more rigorous than in previous approaches. Building a formal model requires a better understanding of the problem domain and a better understanding of the way in which the problem domain is understood. A formal model can explicitly model nondeterminism — when choice of behaviour is specified[1]. Another important feature of a formal method is that high levels of expressibility allow definition of *what* is required without stating *how* it can be achieved. This, together with nondeterminism, supports a powerful freedom of implementation facility.

2.1.4 Formalising the Object Oriented Approach

A formal approach to object oriented analysis requires:

- **A method for gaining understanding** of the problem so that deciding the relevance of any part of the problem domain is straightforward. In other words, a mechanism is needed for

[1] Nondeterminism is not the same as ambiguity. An ambiguous statement is one which can be interpreted in more than one way.

formally identifying the scope of the problem in terms of its component parts, i.e. the classes and objects.

- **A means of recording the relevant information** in a structured and coherent fashion. The notation used for capturing the properties of the model is fundamental to the method[2]. It must be able to reflect the structure of the problem domain whilst also specifying requirements in an implementation independent way.
- **A way of validating the model** against user requirements. Validation must check the model for well defined properties, allow analysts to test their understanding of the problem, and facilitate customer accountability.

Rather than promoting one particular formal method of representing object oriented requirements, section 2.3 identifies the porperties that such a notation is required to express in an elegant and concise way[3]. We stress that the object oriented aspects of the system being analysed must be prominent in the formal representation. Diagrammatic representations of object oriented properties are encouraged provided they have a formal semantics associated with them.

2.2 Analysis: An Overview

2.2.1 Analysis is Problem Domain Understanding

Analysis is the process of maximising *problem domain understanding*. Only through complete understanding can an analyst comprehend the responsiblities of a system. The modelling of these responsiblities is a natural way of expressing system requirements. The modelling process increases understanding. Once the model is sufficiently rich to express all that is needed, then the analysis is complete and design can begin.

The simplest way for an analyst to increase understanding is through interaction with the customer. The customer may be one person, in which case the RCA process is much simplified; however, it is more likely that the customer is a group of clients, each with their own particular needs. These clients may be people, machines, or both. One of the main problems in dealing with a set of customers is that the interelated set of requirements must be incorporated into one coherent framework. Each client must be able to validate his (or her) own needs irrespective of the other clients (unless of course these needs are contradictory). The partitioning of requirements in this way may not be advantageous to designers. An analyst must be able to understand the set of requirements as a whole: the structuring of requirements as a collection of client needs may or may not be recognised in the design, but it is important that such an option is available.

[2]Method is often confused with diagrammatic notation — the obvious reason for this is that the diagrams are often the most visible (and accessible) part of a method.

[3]Although we do not promote one particular language for expressing requirements, the examples do require a concrete syntax. Consequently, we chose to define the example specifications in OO ACT ONE (the formal object oriented analysis language defined in chapter 3).

Interaction with the customer is an example of informal communication. It is an important part of analysis and, although it cannot be formalised, it is possible to add rigour to the process. A well-defined analysis method can help the communication process by reducing the amount of information an analyst needs to assimilate. By stating the type of information that is useful, it is possible to structure the communication process. Effective analysis is dependent on knowing the sort of information that is required, extracting it from the customer, and recording it in some coherent fashion. This chapter is concerned with identifying the type of information that needs to be recorded during analysis, with a view to defining a suitable method of representing this information.

2.2.2 Traditional Analysis Methods and Models

The past two decades has witnessed the establishment of many different analysis techniques. Each technique places different degrees of emphasis on each of the following:

- **Functional decomposition**, which manages complexity in terms of structured functionality.
- **Information modelling**, which helps to structure understanding by imposing a framework based upon the data in the system.
- **State Transition Diagrams**, which place emphasis on the timing and control aspects of complex behaviour.

The underlying principle of each of these approaches is improving understanding through complexity management. Each approach provides one consistent view of a system and its parts. The problem central to analysis is that some approaches are more useful than others in particular problem domains. Since most systems are complex, to various degrees, in three aspects — function, data, and timing — it is hard to see how three different models can be balanced in one coherent method. An object oriented analysis (OOA) method provides a framework in which all three aspects of system behaviour can be represented, although the data provides the basis for the structure. Before OOA is examined, each of the other approaches is examined in more detail.

2.2.2.1 Functional Decomposition

Whilst functional decomposition is a straightforward application of the divide and conquer maxim, it has one fundamental flaw — there is no explicit statement of problem domain understanding. The mapping between functional requirements and the subject matter is often indicative of the way the analyst sees the problem rather than how the customer views it. By emphasising functionality, the need for mutual understanding between customer and analyst is ignored.

Another difficulty with functional decomposition arises when the system being analysed does not appear to be providing a service which can be characterised by one all encompassing function. A system may be meeting a number of different needs whose inter-relationship can be seen only through a thorough understanding of the whole problem domain.

A final problem with a functional approach to decomposition is volatility. It has been argued that the most volatile aspects of system requirements are the functions [84]. Consequently, the structure of

a specification which is based on functionality may not be stable. Stable structure is an important part of the analysis — identifying the most persistent elements of a problem, and basing understanding around them, is fundamental to good requirements capture.

Functional decomposition is not to be totally ignored. It is a useful approach when considering simple systems (or components). When decomposition (not necessarily functional) has produced a component whose behaviour is easily characterised as some function, or group of functions, then any further decomposition should be functional. In an object oriented approach, the services an object provides can be thought of as providing system functionality.

2.2.2.2 Data Flow Models

This modelling technique is fundamental to many of the structured analysis methods first proposed in the 1970's and carried through to the present, for example [90, 94, 49, 27]. System requirements are modelled using data flow models as follows:

> A customer has a need, or set of needs. Each need is modelled as an interaction with the system. An interaction is represented by input and output flows of data at nodes (which can be thought of as data processors). Data stores (which are also represented by nodes) are connected to other nodes in the system. Grouping nodes into subsystems is the way in which structure is added to improve understanding. However, grouping is informal and often arbitrary. Functional decomposition is applied to nodes at the lowest level.

Data flow models are hard to reason about — especially in large systems where the environment interface is complex. The partitioning of nodes is not well understood, and this can result in a data flow model structure which bears no resemblance to the structure of the problem domain. It is difficult to to comprehend the overall behaviour of a reasonably sized data flow model. The difficulties in validating a data flow model against customer requirements are enormous.

2.2.2.3 Information Modelling

Modelling the world in data is most closely related to the way in which humans view their environment. Entity relationship diagrams, semantic data models and information modelling all refer to objects or entities in the real world (see [17] for an overview of these techniques). Every object has an associated set of attributes or properties, and there are ways of relating different objects. In its purest form, information modelling shows only the structure of a problem in terms of the data. There is no explicit statement of functional requirements. Therefore, the responsibilities of a system are not explicitly stated.

2.2.2.4 State Transition Diagrams

State transition diagrams are most useful for modelling systems whose behaviour progresses through different states over time. For example, a person can be modelled as proceeeding through the following

sequence of states — born, child, adolescent, single adult, married adult, widowed adult and dead. This type of behaviour is an important aspect of all systems but it is not clear how other properties can be incorporated in such a simple model. Furthermore, some systems go through a large (potentially infinite) number of states and the structuring of such behaviour is often quite difficult. It is more reasonable to attempt to incorporate the notion of state transitions in a more constructive model. (State transition diagrams, in many different guises, are a useful means of providing an underlying semantics to some other less abstract models. A labelled state transition system plays the semantic role in the definition of OO ACT ONE (the formal object oriented analysis and requirements capture language.)

2.2.2.5 Combining Different Models

Before object oriented analysis, the three approaches to handling complexity were 'thrown together' in different ways in different methods. This gives rise to confusion when a precise statement of how to carry out the analysis is required. In many cases, there is a great deal of arm waving to connect data, function and structure. An object oriented approach provides a much more meaningful way of incorporating function and state attributes in the same model. The separation of function and state is not an issue during object oriented analysis because the lowest-level building blocks (the objects) are defined as combinations of both parts.

2.2.3 Features of Good Analysis

There follows a list of features which should be present in an analysis technique for it to be considered *good*. Each feature is seen to various degrees (or not at all) in each of the afore mentioned modelling approaches. OO ACT ONE, the formal object oriented analysis language proposed in this thesis, is examined in section 4.5.1. as one particular language which facilitates the meeting of these requirements.

A good analysis technique must:

- **Be amenable to changes in the requirements within a stable structure**
  It is important that an analysis method is flexible enough to readily incorporate changes in the requirements. Three types of change must be catered for:

  - Extension: when new requirements are added.
  - Alteration: when old requirements are changed.
  - Re-conceptualisation: when the same requirements are expressed differently.

  These changes must be kept as localised as possible. Central to controlling change is the development of a stable structure upon which behaviour can be specified as a set of distinct though interrelated parts.
- **Encourage Re-use**
  The issue of re-use has been well debated in the programming environment (see [59]). A good analysis method must encourage component and structural re-use. This is one of the areas

in which a formal approach is vital. A more difficult type of re-use to quantify is the notion of experience, when insight is gained into methods of application in different circumstances. Within analysis, the learning of a method should very quickly correspond to gaining experience. Only in the initial learning period should analysts be concerned with notation and concepts: a good analysis method should be based on very simple principles.

- **Act as an interface between customers and designers**
  The analysis model must be capable of fulfilling two very different needs. Firstly, it must be *customer oriented*, i.e. there must be a direct correspondence between the model and how the customer views the problem. Secondly, the model must be useful to designers. The system requirements must be easily extracted, and the structure of the problem domain must be visible for (potential) re-use in the solution domain. The easiest way in which a model can play this dual role is if the same underlying notions and principles are present in the problem and solution spaces.
- **Incorporate standard modelling techniques**
  There are many tried and trusted techniques for complexity management. These are found in many different forms and in many different areas. The five central concepts are:

  - **i) Abstraction**: any mechanism by which irrelevant information can be set aside (perhaps for consideration at a later stage). Functional abstraction is a means of saying what something does without saying how it does it. Data abstraction is a way of specifying state as an interface rather than as contents. (In object oriented approaches, the notion of service further abstracts away from the difference between function and state.) Functional and data abstraction hide detail at a lower level. A different kind of abstraction emphasises detail in one part of the system by hiding information in a different part of the system at the same level.
  - **ii) (De)Composition**: any mechanism by which a system (or component of a system) can be defined as a number of interacting (sub)components. Composition is the grouping together of behaviours to form a more complex behaviour. Decomposition is the realisation of a complex behaviour through division into simpler components. In other words, they are opposite sides of the same coin.
  - **iii) Classification**: a means of classifying components into meaningful hierarchies. The way in which humans think is based on a conceptualisation of complex, often intertwinned, classifications. The ability to group together objects according to shared (common) properties is fundamental to real world understanding. The same can be said of real world modelling.
  - **iv) Communication**: some means of modelling interaction between components. In a complex system it is important that all interaction between components is well defined and clearly understood by customer, analyst and designers.
  - **v) Relationship Co-ordination**: categorisation, composition and communication properties give rise to three different types of relationship, namely *is-a*, *has-a* and *interacts-with*.

It is important that the analyst models these relationships in a consistent and coherent fashion.

- **Have a formal basis**
  Mathematical rigour is necessary for formal validation, testing and completeness and consistency checking. The advantages of formal methods in the specification of requirements are well documented (see [20, 50], for example).

2.2.4 Introducing Object Oriented Analysis

The principle upon which object oriented analysis (OOA) is based is the direct mapping of problem domain entities and responsibilities into a requirements model. The entities (objects) are described in terms of the interface through which they interact with their environment. The services offered at an interface abstract away from the *how* to the *what*. Encapsulation and abstraction, two of the most important modelling techniques, are implicit in an object oriented approach. OOA incorporates all the desirable features (other than the requirement for formality) within a consistent framework of understanding. The structure of problem understanding is the framework upon which the remaining stages of development, namely design and implementation, are based. OOA terminology has arisen from two very different sources:

- **Object Oriented programming languages**
  A programming language is said to be object oriented[4] if it includes the notions of object, encapsulation, message passing between objects, class, inheritance, dynamic binding and polymorphism. The conceptual consistency between the different development stages, which is one of the main advantages of the object oriented approach, is also a disadvantage when the terminology is not clearly defined. Generally, the programming notions of class, object and inheritance are imprecise. It is important that this imprecision is not evident in analysis.
- **Information modelling**
  Information modelling has resulted in a more analysis-like view of objects/entities. However, information models do not facilitate the definition of function or behaviour. Also, the modelling diagrams are informal and open to interpretation. Information modelling is good for representing the structure of the data being considered. It is not good at representing the classification relationships between the data containers.

As a preview of section 2.3, the key concepts of OOA are given below. They are not formally defined and as such only introduce the notions. Object oriented terminology is employed differently in different environments. This is one of the main problems with object oriented methods. A major part of the development of a FOOA technique is the provision of well-defined meaning to the concepts. The informal list of terminology, below, illustrates the problems — one person's class is another person's object!

[4] [124] defines three catagories of 'object language', namely object oriented, class oriented and object based.

2.2.5 Objects and Classes: The Problems with Terminology

For each key concept, a number of definitions are given. Each of the definitions is 'correct' in its own particular context. This illustrates the confusion that exists in object oriented terminology.

- **Objects:**
  - An object is anything which can be uniquely identified.
  - An object is an entity which plays some role in the behaviour of the system under consideration.
  - An object is some thing which encapsulates state, and the set of operations on that state.
  - Objects are instances of abstract data types.
  - An object is an element of a particular set (or class).
- **Classes:**
  - A class is a collection of objects.
  - A class is a set of related behaviours.
  - A class is a type.
  - A class is a description of properties common to a set of objects.
  - A class describes an implementation, or group of implementations, of an abstract data type.
- **Inheritance:**
  - Inheritance is a means of representing relationships between classes.
  - Inheritance is a subtyping relation.
  - Inheritance is an incremental code modification technique.
  - Inheritance is a means of defining a class as a modification of one (or more) other classes.
  - Inheritance is a code re-use facility.
  - Inheritance is a tool for conceptually grouping together sets of behaviours with some properties in common.
  - Inheritance is a tool for enforcing properties between instances of different classes.
- **Object interaction:**
  - Objects interact by passing messages to each other.
  - Object interaction is through a well defined interface.
  - A service is provided by one object when it is asked to do something by another object.
- **(De)Composition:**
  - A class can be defined as some sort of composition of two (or more) other classes.
  - An object can be realised as a composition of instances of two (or more) interacting objects.
  - (De)composition is a code re-use facility.
  - (De)composition is a structuring mechanism.

The statements above emphasise the informal (and sometimes inconsistent) use of object oriented terminology. A FOOA method must remove this informality by defining each of the terms in a clear, concise and unambiguous way.

2.3 Object Oriented Analysis: An Informal Approach

This section does not continue the promotion of OOA through an extension of the list of well documented object oriented 'blessings'. The three main undisputed features of the object oriented paradigm are:

- Consistency of method and notation throughout development.
- Modelling of the problem as it is viewed in the real world.
- Inherent abstraction and encapsulation.

These are the foundations upon which claims for extensibility, re-usability, improved understanding, and maintainability are built. Rather than elaborating on the object oriented claims, this section makes the assumption that object orientation provides the basis of a good approach to requirements capture and analysis. In this way, the crux of OOA can finally be considered: *how?* rather than *why?*

Sections 2.3.1, 2.3.2 and 2.3.3 give an informal introduction to the notions of objects and class, and the relationships between them. This gives rise to a number of other important issues which are best considered by adopting a particular language for the expression of object oriented requirements.

In section 2.3.4, many examples which are used in which a concrete syntax for recording object oriented properties of a system is introduced. This serves two purposes. Firstly, it introduces the concepts and relationships that are important in object oriented analysis. Secondly, it provides a means of examining the requirements that an analyst is likely to place on a formal object oriented language.

2.3.1 Identifying Objects

At first glance, the notion of object seems to be the key to the object oriented approach. It is important that our intuitive feel for what makes an object an object is reflected in a formal definition. To stimulate thought, we list a wide variety of *things* that could be considered to be valid objects. These *things* may have been identified during the analysis of a number of systems. We attempt to identify their 'objectness', i.e. the features that they share in common.

- **People:** You, me, John Major, the British Prime Minister, my mother, etc.
- **Structures:** The Eiffel Tower, Edinburgh Castle, the M25, your bank, the house you last slept in, etc.
- **Places:** Europe, Australia, Stirling University, Paris, the Eiffel Tower, your bank, the bank manager's office in your bank, etc.
- **Machines:** The watch on your wrist[5], the computer on which this was written, my car, etc.
- **Systems:** The computer on which this was written, the M25, the BT telephone network, your bank, etc.
- **Events:** The second world war, your birth, the last world chess championships, the 1992 Olympics, etc.

[5]If there is one — a watch that is!

- **Concepts (Abstractions):** Chess, the English language, the number 6, the Greek letter Π, the mathematical constant Π, etc.
- **Classifications:** people, structures, cars, trains, beds, houses, games, trees, songs, tunes, computers, planets, number systems, etc.

The first thing to notice is that there is always an informal link between the label (in this case a string of characters) and the object which the label identifies. In fact, several different labels can be applied to the same object. Worse still is the fact that one label can be applied to two different objects — John Major is not always a reference to the British Prime Minister. A major[6] difficulty is that the context in which the label is used, and the assumptions made by the reader, are fundamental in the mapping between label and object. It is clear that an analyst cannot work with the actual objects and so the labels act as abstractions for the objects. A **label** is a form of identification. This leads to the first property which must be fulfilled by an object: it must be uniquely identifiable by its label within the context of the problem domain.

An **attribute** is some property of an object which plays a part in it being uniquely identified. For example, an attribute of Paris is that it is the capital of France. A different attribute is that it is the city in which the Eiffel Tower is found. A rather different set of attributes may include the latitude and longitude of the city centre, a list of all the street names, or even a list of all the people in the Paris telephone directory. Each of these attributes is sufficient to identify 'Paris'; but, it is not clear if this 'Paris' is a city, a part of a map, or a tourist centre. **Abstraction** is the means by which only the relevant attributes of some entity are considered. In different systems, and different problem domains, one object may have different model abstractions. To continue with Paris as an example: in a model of the globe, the latitude and longitude are important but, in a model of the telephone network they do not play an obvious role. This leads to the second property of an object: it must have some well defined set of **attributes**.

An object must be **encapsulated** so that its internal mechanisms are used only through some well defined **interface**. We define the **external attributes** of an object as precisely those feature which can be accessed in this way. A class of objects is said to provide a set of **services**. Each **service** corresponds to an **external attribute**. A **service request** to an object (at its external interface) is a means of invoking some response. The object providing the service (the **service provider**) may change its internal state, or output some 'result', or a combination of both.

It is evident that an object should be characterised by its external attributes, i.e. the operations which are serviced by its interface, if its representation is to be *implementation independent*. The internal structure, reflecting function and state, is the means by which the external attributes are defined. It is often necessary to structure the statement of requirements to aid understanding. Further, structure is necessary to define behaviour of objects which can attain infinitely many different states. Structure influences the implementation process. However, constructive specifications do not necessarily impose implementation decisions on designers. The structure of problem domain understanding is recorded during object oriented analysis. Implementers (and designers) are not obliged to

[6] No pun intended.

use the structure of the analysis in the solution domain but, in some cases, this re-use of structure is beneficial.

In conclusion, the definition of an object must incorporate:

- A means of identification.
- An interface which encapsulates the object by forcing access to the object to be through a well defined set of external attributes (services).
- The 'meaning' of the external behaviour, i.e. a statement of how an object responds to service requests.
- An internal state, i.e. a mechanism such that an object, as a dynamic entity, is able to progress through a sequence of states depending on its interaction with its external environment.

Two objects belong to the same **class**(ification) when they exhibit 'the same behaviour' through their interfaces[7]. Informally, they must offer the same set of external attributes, and the way in which these attributes are fulfilled must be the same when two objects in the same class have the same internal state. In object oriented terminology, a **class** embodies the concept of a set of objects together with some common behaviour characterised by a set of external attributes.

2.3.2 Identifying Classes

A class is a set of member objects offering common behaviour. A class definition must contain the following:

- A list of external attributes which all member objects must provide.
- A means of identifying member objects.
- A semantics defining the behaviour resulting from the servicing of external attributes for every member object of a class.

For example, Range1to9 can be defined as follows:

- The external attributes are addition, subtraction and equality.
- The members are the integers 1, 2, 3, ..., 9.
- The semantics of the service methods associated with the three external attributes are those normally associated with integers.

We chose to think of a class as a parameterised set of behaviour. Each member of a class is identified by one particular realisation of the behaviour parameters. An object refers to one particular class member at any instance in its lifetime. Thus, a dynamic object references a sequence of class members as it progresses through a sequence of states prompted by interactions with its environment.

Identifying classes in the problem domain is fundamental to OOA. Classification provides us with a basis on which a framework of understanding and representation can be built. Moreover, human understanding has evolved through our ability to classify and categorise to various levels of abstraction. Therefore, it is reasonable to assume that classification must play a leading role in analysis.

[7] This notion of class membership is formally defined in chapter 3

2.3.3 Classification Relationships

By studying relationships between objects, between classes, and between classes and objects, it is possible to gain a better understanding of these concepts with relation to analysis and requirements capture. The two most fundamental relationships are class membership and subclassing:
Class Membership: When an object a is said to be a member of a class A we write $a \in A$.
Subclassing: When a class A is said to be a subclass of a class B we write $A \sqsubseteq B$.

It is possible for a class to be a subclass of more than one superclass[8]. For example, a cat class is a subclass of the class of mammals, and a cat class is also a subclass of the class of pets[9]. Note that a cat is classified by two different types of attributes: the physical attribute of being a mammal, and the functional attribute of being a pet. Analysis of most systems gives rise to the identification of objects whose relevant set of external attributes is different in different contexts. Distinguishing between different categories of attribute increases understanding of the problem domain, and this understanding can be represented diagramatically in a **class hierarchy**. Class hierarchies provide a fundamental way of structuring object oriented requirements.

A simple way to structure the analysis of a problem is to first identify the class hierarchies. These structures show one type of relationship between entities in the problem domain, namely subclassing. Identifying classes and putting them into a coherent framework is fundamental to formal object oriented analysis. A simple example (a hall of residence) is examined below. It illustrates the power of an approach in which classification is the main form of analysis. It also focuses attention on the limitations imposed by restricting analysis to the identification of subclassing relationships.

Classification Example: A Hall of Residence

A decision has been made to computerise the records for the halls of residence in a university. In particular, one part of the system is concerned with the residents. Analysis of the residents has led to the following classifications:

- Residents are either students or non-students.
- Residents are either staff or non-staff.
- Staff are either wardens or porters.
- Wardens are either subwardens or hallwardens.
- Subwardens are students.
- Hallwardens are non-students.
- Ordinary residents are non-staff and students.

These relationships are shown in the class hierarchy in figure 2.1.

In this example, the class of `Subwardens` is a subclass of `Wardens`, and a subclass of `Students`. This multiple classification is very powerful within analysis. A subwarden can be regarded as a member of staff in one context, and a student in a different context.

[8] $A \sqsubseteq B \Leftrightarrow A$ is a subclass of $B \Leftrightarrow B$ is a superclass of A.

[9] In this case it is questionable whether such a simple classification is appropriate — petting tigers is not recommended!

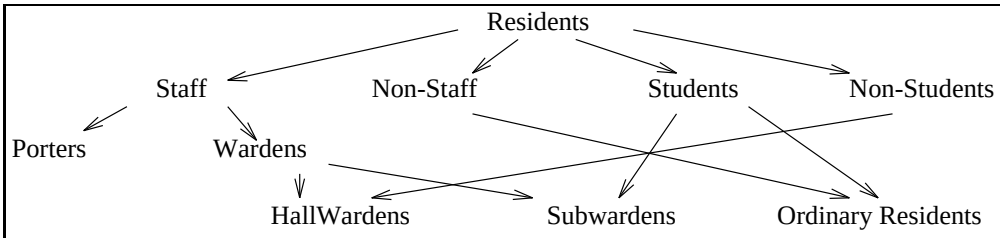


Figure 2.1: A Hall Residents Class Hierarchy

Subclassing : The Limitations

Restricting analysis to the identification of class relationships has a number of limitations. In the analysis of the hall residents system, classification does not:

- Provide a means of recording the number of class members from each class in the residences.
- Consider the functional aspects of the system. (Although none of responsibilities of the system are shown, the class hierarchy does provide a structure upon which the functional aspects can be decomposed.)
- Represent the communication, synchronisation, or timing aspects of the system.

The first limitation is overcome through the introduction of a different type of relationship, namely composition. The functional aspects of a system are represented when classes are defined in terms of their external attributes. Timing and synchronisation aspects of analysis are more difficult to map directly onto the object oriented framework: they are considered in sections 4.2. and 4.3.

2.3.4 Defining Classes of Behaviour

This section introduces the concepts central to recording object oriented requirements during analysis. A number of simple object oriented behaviours are considered. A concrete syntax for the specification of object oriented requirements, namely OO ACT ONE, is introduced. OO ACT ONE is formally defined in chapter 3: its use at this stage of the thesis is intuitive and requires no knowledge of the underlying formality.

2.3.4.1 LITERALS: an explicit identification of class members

The simplest object oriented property to identify and specify must be class membership. In OO ACT ONE, a literal is a label (defined as a sequence of characters) which uniquely identifies one member of a class. Consider the class `ComparisonResult` defined in example 1.

```
(* Example 1:  LITERALS *)
CLASS ComparisonResult OPNS
LITERALS: before, after, same
ENDCLASS (* ComparisonResult *)
```

The specification defines the `ComparisonResult` class to have three members: `before, after` and `same`. There are no external attributes offered by this class (it can be used only as a passive carrier of data).

2.3.4.2 STRUCTURES: parameterising the specification of class membership

The members of a class represent the set of states that an object can attain. It is necessary to extend the `LITERAL` concept to enable the specification of a set of class members in a parameterised fashion since:

- A class with a large number of literal members is unwieldy and inconcise.
- Parameterisation of class members adds structure to the specification and improves understanding.
- Parameterisation is the only means of defining classes of behaviour with an infinite number of members.

These points are reinforced by examples 2 and 3, which follow.

```
(* Example 2:  STRUCTURES for ease of expression *)
CLASS C-RPair USING ComparisonResult OPNS
STRUCTURES: pair<ComparisonResult, ComparisonResult>
ENDCLASS (* CR-Pair *)
```

The set of `C-RPair` class members is not defined explicitly as a list of literal values. Rather, a `STRUCTURE` operation is used to parameterise the specification of class members. The set of class members can be realised through instantiation of the `STRUCTURE` parameters:
{`pair(before,before), pair(before,after), pair(before,same), pair(after,before), pair(after,after), pair(after,same), pair(same,before), pair(same,after), pair(same,same)`}.

In this example, the set of nine class members can be defined using nine `LITERALS`. However, the `STRUCTURE` definition is more concise and the labelling of the `STRUCTURE` operation as a `pair` improves the specification. In example 3, the `Number` class has an infinite set of members: {`0, succ(0), succ(succ(0)), ...`}. Classes with infinite behaviours arise in two different ways:

- Recursive `STRUCTURE` definitions define a `STRUCTURE` operation to have a parameter whose class is the same as the class in which the `STRUCTURE` is defined (see example 3).
- Non-recursive `STRUCTURE` definitions define an infinite class when one of the component classes is infinite.

```
(* Example 3:  STRUCTURES for specifying infinite classes*)
CLASS Number OPNS
LITERALS: 0
STRUCTURES: succ<Number>
ENDCLASS (* Number *)
```

2.3.4.3 ACCESSORS: external attributes for getting responses from objects

Examples 1 to 3 define only classes as sets of passive members. The objects in each of these classes do not offer external attributes. We require a means of defining objects which can be asked to perform a service through their external interface. One such service an object can provide is to return some information about itself. Example 4 illustrates the specification of three `ACCESSOR` attributes, i.e. attributes which give *access to* some internal details of the object servicing the request.

```
(* Example 4:  Unparameterised ACCESSOR operations *)
CLASS ThreeOrderA USING ComparisonResult OPNS
LITERALS:1,2,3
ACCESSORS: compare1 -> ComparisonResult, compare2 -> ComparisonResult,
compare3 -> ComparisonResult
EQNS
1..compare1 = same; 1..compare2 = before; 1..compare3 = before;
2..compare1 = after; 2..compare2 = same; 2..compare3 = before;
3..compare1 = after; 3..compare2 = after; 3..compare3 = same
ENDCLASS (* ThreeOrderA *)
```

An object of the class `ThreeOrderA` has state corresponding to one of the class members `1,2` or `3`. Such an object offers three external `ACCESSOR` attributes, namely `compare1, compare2` and `compare3`. Servicing an `ACCESSOR` results in the object servicing the request returning some value (a member of the class specified after the right arrow in the `ACCESSOR` operation definition). The object does not change its internal state.

The way in which each object of a class responds to an `ACCESSOR` request must be defined as part of the OO ACT ONE specification. `ACCESSOR` equations are defined in an expression of the form: `obj..accessor = ...`, where `obj` is a class member, `accessor` is the name of an `ACCESSOR` operation and the right hand side of the equation represents the result returned when an object with state `obj` services `accessor`.

```
(* Example 5:  Parameterised ACCESSORS *)
CLASS ThreeOrderB USING ComparisonResult OPNS
LITERALS:1,2,3
ACCESSORS: compare<ThreeOrderB> -> ComparisonResult
EQNS
1..compare(1) = same; 1..compare(2) = before; 1..compare(3) = before;
2..compare(1) = after; 2..compare(2) = same; 2..compare(3) = before;
3..compare(1) = after; 3..compare(2) = after; 3..compare(3) = same
ENDCLASS (* ThreeOrderB *)
```

The `ThreeOrderB` class, in example 5, shows how `ACCESSOR` operations can be parameterised. In this case, `compare` is parameterised by class `ThreeOrderB`. The parameterisation of the external attributes of a class is necessary if we wish to model the service requester providing 'input parameter values' to the service provider.

2.3.4.4 TRANSFORMERS: defining external attributes for history dependent behaviour

The behaviours of the `ThreeOrder` classes are history independent: they have no external attributes which change the state of the object servicing the request. Such classes are similar to types in imperative programming languages. The class members are analogous to constant values.

The behaviour of most objects depends on the previous services which they have 'carried out'. One of the most common attributes of an object is the ability to accept some new information, remember it, and use it in response to a later request. In an object oriented analysis language it is necessary that history dependent behaviour can be defined. In OO ACT ONE, the simplest example of this type of behaviour corresponds to the imperative notion of a variable (given below in example 6).

```
(* Example 6:  TRANSFORMERS: specifying history dependent behaviour *)
CLASS int-var USING integer OPNS
STRUCTURES: an-integer<integer>
TRANSFORMERS: update<integer>
ACCESSORS: recall -> integer
EQNS
an-integer(integer1)..recall = integer1; an-integer(integer1).update(integer2) =
an-integer(integer2)
ENDCLASS (* int-var *).
```

The `update TRANSFORMER` operation of the `int-var` class changes the internal state of the object servicing the request. The object does not return any value to the service requester. Like `ACCESSORS`, `TRANSFORMER` operations can be parameterised. The new state of an object after servicing a `TRANSFORMER` operation is defined by an expression of the following form: `obj.transformer = ....`

2.3.4.5 Parameterised Structure Equation Definitions

Example 6 illustrates the need to be able to define infinite behaviour in a parameterised form. Given a class `integer` defined to have an infinite number of members, class `int-var` also has an infinite number of members. The behaviour associated with each of these members must be defined in the equation body of the class. This is possible only through some form of equation parameterisation.

For example, `an-integer(integer1)..recall = integer1;` specifies that for every `integer1` which is a member of class `integer`, an object of class `int-var` with state `an-integer(integer1)` returns `integer1` in response to a `recall` service request. Similarly, the `TRANSFORMER` behaviour is also defined in a parameterised fashion.

Note that it is the variable parameters `integer1` and `integer2` are used to define the behaviour of `int-var` in a parameterised fashion. In our notation, the class of a variable parameter is identified by the string of characters which precede the last numeric character(s) of the variable identifier. All variable parameters in equation definitions must be represented in this way[10]. Equations which are

[10] An advantage of this approach is that the variable identifier also identifies the class to which the variable belongs. The disadvantage of variable names not describing their function is negated by using comments when it is necessary to

```
(* Example 7:  Dynamic Structure *)
CLASS list USING integer,Bool OPNS
LITERALS: empty
STRUCTURES: S-list<list,integer>
ACCESSORS: check<integer> -> Bool
TRANSFORMERS: store<integer>
EQNS
empty..check(integer1) = false;
S-list(list1, integer1)..check(integer2) = (integer1..eq(integer2))..or(list1..check(integer2));
list1.store(integer1) = S-list(list1, integer1)
ENDCLASS (* list *)
```

parameterised on variable parameters are, by definition, true for all values of these variables.

2.3.4.6 Dynamic Structure

The `int-var` example shows how to record the attributes of an object which has constant state structure. The state is said to be constant because all the data fields are fixed at creation (although the values in the fields may not be fixed) by one structure operation. In example 7 we specify a behaviour which does not exhibit static internal structure.

The `list` class is a simple store of integers which has two external attributes: `store` and `check`. The `store` attribute is used to put integer values into the receiving object of the `list` class. The `check` attribute is used to test if a given integer value has been previously `stored`. The `S-List` operation is said to define a recursive structure.

2.3.4.7 Dependencies Between Classes

Many of the previous examples have class headers of the following form: `CLASS A USING B,...OPNS`. It is a requirement of an anlaysis language that pre-defined behaviours can be re-used in the specification of new behaviour. The `USING` construct provides the basis for such re-use. Example 7 is the first specification in which the classes `used` are not just passive data carriers but actually play a fundamental role in the behaviour of the new class being specified. The `check` attribute of class `list` makes comparisons between `integers` which have been `stored` and an input `integer` parameter. This comparison depends on the following behaviour being defined in the classes used by `list`:

- Class `integer` must have an external ACCESSOR attribute defined as `eq<integer> -> Bool`.
- Class `Bool` must have an external ACCESSOR attribute defined as `or<Bool> -> Bool`.

2.3.4.8 Multiple Structure Operations

In object oriented analysis we may identify a class of behaviour which is made up of 2, or more, distinct groups. For example, students at university may be either single honours students or joint honours students. Although the external interface of these two groups must be the same when they

say what role a particular variable takes.

are part of the same class, we require a mechanism to distinguish between them. Such a mechanism already exists, namely STRUCTURE operations. In OO ACT ONE we record this type of behaviour as a class with more than one STRUCTURE operation (see example 8).

```
(* Example 8:  Multiple Structures *)
CLASS Students USING Subject, Bool OPNS
STRUCTURES: Single<Subject>, Joint<Subject,Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
Single(Subject1)..Studies(Subject2) = Subject1..eq(Subject2);
Joint(Subject1,Subject2)..Studies(Subject3) =
(Subject1..eq(Subject3))..or(Subject2..eq(Subject3))
ENDCLASS (* Students *)
```

The STRUCTURE mechanism provides a natural way of describing this type of class partitioning. Note that a TRANSFORMER operation can be defined to model a single honours student changing to be a joint honours student (or vice-versa). Multiple STRUCTURE operations can be used to model objects with dynamic structure.

2.3.4.9 DUALS: A means of combining ACCESSORS and TRANSFORMERS

The previous examples have defined classes with two different types of attribute — ACCESSORS and TRANSFORMERS. We identify the need for an attribute which is a combination of these. For example, a stack class may store integers. An attribute pop is required to model the removal of an integer from the stack (i.e. a change of state) and the return of this information to the pop requester. To model this type of service in OO ACT ONE, a DUAL attribute is defined. This is illustrated in example 9.

```
(* Example 9:  DUAL attributes *)
CLASS stack USING integer, Bool OPNS
LITERALS: empty
STRUCTURES: Sstack<stack, integer>
DUALS: pop -> integer
TRANSFORMERS: push<integer>
EQNS
empty.pop = empty AND ~integer;
Sstack(stack1, integer1).pop = stack1 AND integer1;
stack1.push(integer1) = Sstack(stack1, integer1)
ENDCLASS (* stack *)
```

A DUAL equation is defined as the conjunction of an ACCESSOR equation and a TRANSFORMER equation. For example, `empty.pop = empty AND ~integer;` specifies that an empty stack 'changes' state to being empty in response to a pop request and returns the value ~integer to the service requester.

Modelling DUAL behaviour is fundamental to object oriented analysis and requirements capture. It can be argued that such behaviour can be adequately represented by an ACCESSOR followed im-

mediately by a TRANSFORMER. However, this dual model depends on some 'lock out' facility between servicing the ACCESSOR and the TRANSFORMER. Such a facility is implementation oriented and as such does not provide a good model for analysing this type of behaviour. The DUAL mechanism abstracts away from the *how* to the *what*.

2.3.4.10 Unspecified Class Members: handling exceptions

Example 9 illustrates the first instance of explicitly defining behaviour of a class to be unspecified. An empty stack cannot return a meaningful result in response to a pop request. Certainly, there are a number of different options for coping with such exceptions, but an analysis language must handle them in as abstract a way as possible. In our object oriented model it is necessary that pop is defined for the member empty, otherwise empty would not be a valid stack. Rather than adopting a particular implementation strategy to deal with exceptions (like 'just return a 0') we define a mechanism to enable analysts to defer exception handling to the designers and implementers.

In OO ACT ONE, all classes are defined to have an unspecified literal member, represented by the class name preceded by a '$\sim$' character. This member is implicit in every class specification and is used to represent behaviour which the analyst may not wish to specify at this stage of development. By default, the external attributes of unspecified members are defined to result in unspecified behaviour of the appropriate class (see chapter 3).

2.3.4.11 Distinguishing Between Accessors and Transformers

Example 10 is included to emphasise the importance of distinguishing between TRANSFORMERS and ACCESSORS. In some object oriented models this is not done (for example, see [14]). We define a linked list of integers (Linked-List) with transformer and accessor operations which seem to define identical behaviour. These operations are tailT and tailA. However, our object oriented interpretation of the behaviours offered by these attributes is very different. A Linked-List object, in response to a tailT request, updates its internal state by removing the last integer element which was added. It does not return any result to the service requester. Contrastingly, the same object, in response to a tailA request does not update its internal state, but it does return a result to the service provider. Ambiguous specifications arise if TRANSFORMERS and ACCESSORS are not distinguished.

2.3.4.12 Invariant Properties

A class invariant is some property which every member of that class must fulfil. We require an object oriented analysis language to incorporate some sort of invariant mechanism.

OO ACT ONE provides two types of invariant mechanism: class invariants and structure invariants. These are illustrated by examples 11 and 12. Again, the precise meaning of these invariants is unimportant at this stage: it is the principle behind invariant properties which is important.

Class StudentsB is similar to class Students (see example 8) except that there is an additional invariant property which guarantees that a joint honours student studies two different subjects.

```
(* Example 10:  Distinguishing between accessors and transformers *)
CLASS Linked-List USING integer, Bool OPNS
LITERALS: empty STRUCTURES: S-Linked-List<Linked-List,integer>
ACCESSORS: tailA -> Linked-List
TRANSFORMERS: add<integer>, tailT
EQNS
empty..tailA = ∼list;
S-Linked-List(Linked-List1, integer1)..tailA = Linked-List1;
empty.tailT = ∼list;
S-Linked-List(Linked-List1, integer1).tailT = Linked-List1;
Linked-List1.add(integer1) = S-Linked-List(Linked-List1, integer1)
ENDCLASS (* Linked-List *)
```

```
(* Example 11:  Structure invariants *)
CLASS StudentsB USING Subject, Bool OPNS
STRUCTURES: Singles<Subject>, Joints<Subject, Subject >
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: Joint(Subject1, Subject2) REQUIRES Subject1..neq(Subject2)
EQNS ...
ENDCLASS (* StudentsB *)
```

The class `MathsStudents` is constructed from the members of `Students` which study `Maths`. Chapter 3, section 3.4, examines the OO ACT ONE invariant mechanisms in much more detail.

2.3.4.13 Composition vs Subclassing: Introducing the Problem

Often, object oriented programmers use inheritance (a subclassing mechanism) as a code sharing technique rather than in recognition of an actual subclassing relationship between classes (Meyer [84] often uses inheritance in this way and Stein [104] argues that delegation is inheritance). This is problematic in all areas of object oriented development. The following `vector` example illustrates the problem from two different points of view.

A vector Class

Two different views are as follows:

- **Subclassing** — A vector can be defined to be a subclass of both a magnitude class and a direction class. A vector incorporates all the attributes of a magnitude and all the attributes of a direction. Consequently, a vector is both these things. Subclassing is a natural way of representing these behaviour characteristics.

```
(* Example 12:  class invariants *)
CLASS MathsStudents USING Subject, Bool OPNS
STRUCTURES: Singles<Subject>, Joints<Subject, Subject >
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: MathsSudents1..studies(Maths);
EQNS ...
ENDCLASS (* MathsStudents *)
```

- **Composition** — A vector is constructed from two components, namely a magnitude and a direction. A vector is not a magnitude, it is not a direction, it is some almalgamation of both behaviours into a new class of behaviour.

Depending on which object oriented method is being applied, either of these views is likely to be modelled during the analysis of vector behaviour. During object oriented analysis we must always ask which model is a true reflection of the customer's understanding of the behaviour being specified?

In example 13 we chose to define a `vector` class in a compositional fashion. Every member of the `vector` class is represented by the parameterised structure expression `a-vector(magnitude1, direction1)`. The `a-vector` operation is the only structure of the `vector` class. Consequently, we interpret this to mean that every `vector` object is composed from two component objects (of type `magnitude` and `direction`). We also say, without risk of ambiguity, that a `vector` class is composed from a `magnitude` class and a `direction` class.

```
(* Example 13:  Composition is not subclassing *)
CLASS vector USING magnitude, direction OPNS
STRUCTURES: a-vector<magnitude, direction>
ACCESSORS: length -> magnitude, angle -> direction
TRANSFORMERS: newlength <magnitude>, newangle <direction>
EQNS
a-vector(magnitude1, direction1)..length = magnitude1;
a-vector(magnitude1, direction1)..angle = direction1;
a-vector(magnitude1, direction1).newlength(magnitude2) = a-vector(magnitude2, direction1);
a-vector(magnitude1, direction1).newangle(direction2) = a-vector(magnitude1, direction2)
ENDCLASS (* vector *)
```

2.3.4.14 Structure and Implementation Independence

This representation of a vector is not the only way of expressing its external behaviour. It is possible to define a `vector` using Cartesian co-ordinates x and y, say. Then, the length can be calculated as $\sqrt{(x^2 + y^2)}$, and the angle can be calculated as $\tan^{-1}(\frac{y}{x})$. This Cartesian representation, rather than the polar form given earlier, is more appropriate when the 'addition' of vectors is prominent in the analysis. However, it is much easier to 'multiply' vectors in polar form. The structure of conceptualisation on which the vector class definition is based can thus be seen to be important with respect to possible extensions to the external attributes.

An analyst must chose one representation over the other. We must question whether it is useful to say that a vector is composed of a magnitude and direction when it is equally likely that it is identified during the analysis as a co-ordinate in Cartesian space? The way in which an analyst views a problem is reflected in the requirements specification. Of necessity, it seems that analysis cannot avoid a predisposition in the view that is presented of the problem in the statement of requirements. It is wrong to ask an analyst to represent all possible conceptualisations — it would result in overly complex specifications without guaranteeing that all 'reasonable' ways of viewing the behaviour had

been recorded. An analyst must always chose the representation which is the best reflection of the way in which the customer understands the behaviour being specified.

2.3.5 Explicit Subclassing Relationships

2.3.5.1 Implicit vs Explicit Relationships

Before considering subclassing in our object oriented analysis models, it is necessary to make a statement concerning implicit and explicit subclassing relationships. In a large, complex system with many classes, irrespective of the precise nature of the subclassing definition, it is probable that there are a large number of subclassing relationships between classes. We distinguish between two different types of relation:

- **Implicit**
  An implicit subclassing relationship is one which has no relevance in the specification. For example, consider a `wine` class which has two `ACCESSOR` attributes: `name` and `year`, and a `person` class which has three `ACCESSOR` attributes `name`, `year` and `age`. Depending on how these classes are defined, it is possible that `person` is a subclass of `wine`. In other words, all members of the `person` class are also members of the `wine` class. The consequences of this in an implementation of such a model are unthinkable! This type of relationship is referred to as **implicit** since it is inherent in the specification, but is not explicitly acknowledged or used.
- **Explicit**
  An explicit subclassing relationship is one which is explicitly acknowledged within an object oriented specification. For example, if our analysis identifies that all maths students are students then the subclassing relationship $\texttt{MathsStudent} \sqsubseteq \texttt{Students}$ should be recorded explicitly. Consequently, in object oriented analysis we require a mechanism for making such statements and for verifying that the relationship is well-defined. There are two approaches to this problem:
  - i) Define classes in the normal fashion and separately include a list of subclassing relationships which are relevant in the specification.
  - ii) Define explicit classification mechanisms for defining a new class to be a subclass (or superclass) of an already existing class.

In OO ACT ONE we chose the second approach because the explicit classification mechanisms can be defined in a way that guarantees a valid class relationship between the new and old class. The first approach requires a general mechanism for checking subclassing relationships between any two classes. The analysis behind such a mechanism is much more difficult to formulate than that which guarantees subclassing in specific cases. Also, we argue that there are advantages in having a limited number of subclassing primitives.

2.3.5.2 Specialisation and Generalisation

Specialisation of a class's behaviour is a straightforward reduction in the number of member objects through the addition of some property which must be fulfilled by members of the new subclass but

which may not be fulfilled by every member of the original superclass. The new class members continue to provide the corresponding external behaviour as the corresponding members in the old class. The old (super)class behaviour can be said to contain the new (sub)class behaviour. For example, the even integers are a specialisation of the integers.

Generalisation is the inverse of specialisation. For example, the class of integers is a generalisation of the class of even integers and the class of odd integers.

2.3.5.3 Extension and Restriction

Extension involves an addition of new attributes to an already existing class of behaviour. For example, a queue which can be reset to empty is an extension of a queue which cannot be reset.

Restriction is defined as the inverse of extension. Rather than extending an existing class with new attributes, restriction defines a subset of attributes in the existing (sub)class which are offered by the new (super)class.

2.3.5.4 Subclassing: a look ahead

The four explicit class relationships are formally defined in chapter 3, together with a mechanism for combining specialisation and extension. By their nature the explicit class relationships are difficult to analyse using informal examples and thus it is necessary to develop a formal framework for modelling object oriented requirements before we can pursue a rigorous formulation. There are many types of subclassing but, in our formal object oriented analysis, specialisation and extension (and their inverses) are the only two relationships which are deemed a necessary part of an object oriented analysis language.

2.3.6 Reviewing Object Oriented Analysis Language Requirements: A Five Model Approach

2.3.6.1 Five Object Oriented Models

We propose that object oriented methods are dependent on five central relationships. These are as follows:

- **Classification**
  This is a relationship between an object and a class. All objects in a system are classified. Classes correspond to a group of objects which share a particular classification. Classification is fundamental to human understanding.
- **Subclassing**
  This is a relationship between classes. If A is a subclass of B then all members (objects) of A are also members of B. The subclassing relationship is also prominent in human understanding.
- **Composition**
  This is a relationship between objects. One object is said to be composed from its component parts (themselves objects). The classification of an object's components and the relationships

between these components define the internal structure of an object. Often, all the objects in a class exhibit the same internal structure. In this case a class can be said, without risk of ambiguity, to be composed from its component classes. Class composition is a concise way of defining a set of object composition relationships which hold for every class member.

- **Configuration**
  This is also a relationship between objects. Two objects which are components of the same containing object may, or may not, be 'connected together'. When two objects are connected we say that they configure. More specifically, there is some link between their external interfaces. The car example in figure 2.2 helps to clarify this notion.
- **Interaction**
  Interaction is the only dynamic relationship considered during formal object oriented analysis. All the previous four relationships make up a static view of an object oriented system. An interaction represents an event (and the consequences of the event) that occurs in the lifetime of an object oriented system. Interactions occur between objects which have been configured. Note that two objects which are configured do not necessarily interact in the lifetime of the object. The external interactions of a system are defined as those which occur between the system and its environment. Internal interactions occur between the components of a system. Object oriented behaviour defines possible sequences of interactions between an object and its environment.

2.3.6.2 A Five-model Example

Object oriented analysis is the identification of objects in a system, and the subsequent modelling of these five relationships. For example, consider a system which is very well understood, namely the behaviour of a car. The five relationships are illustrated in figure 2.2.

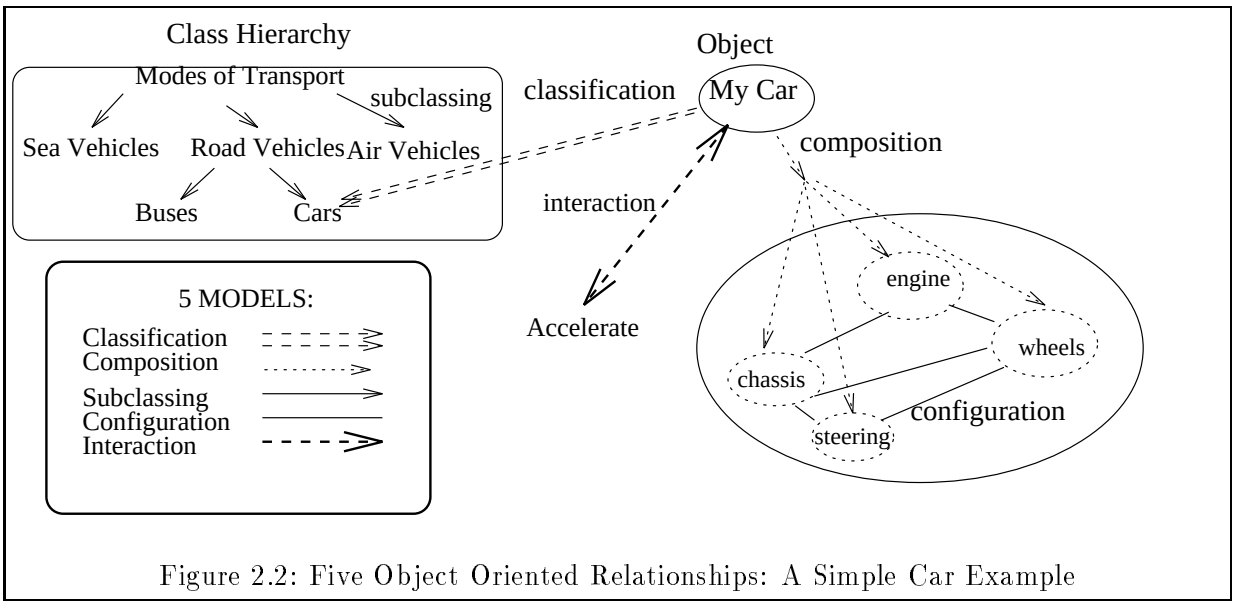


Figure 2.2: Five Object Oriented Relationships: A Simple Car Example

Distinguishing between the five relationships in this analysis is fundamental to human understanding and recording such relationships is therefore crucial in object oriented analysis and requirements capture. These relationships are the basis for the modelling techniques advocated within our object oriented development strategy. Using these models ensures that our object oriented approach is *customer oriented*.

2.3.6.3 FOOA: a review of our model requirements

This section has identified a number of aspects which must be present in a FOOA notation. These are:

- A means of defining classes of parameterised behaviour in terms of an abstract interface.
- A means of uniquely referencing each instance of such a parameterised class.
- A means of modelling a dynamic object as it changes its behaviour over time.
- A subclassing relationship between classes of behaviour.
- A composition facility for defining behaviours in terms of component behaviours.
- A facility for differentiating between three types of external attribute — accessors, transformers and duals.
- An interpretation of the internal structure of an object.

All these requirements are met in an approach based on the generation of the five object oriented models defined above.

2.4 Formal Object Oriented Analysis Using Abstract Data Types (ADTs)

2.4.1 Background to Abstract Data Types

Abstract concepts can be represented in a number of different ways. The means of representation, which is often refered to as the notation, is arbitrary in the sense that there are an infinite number of ways (syntactically) of labelling entities and representing the relationships between them. Natural languages illustrate the diverse range of notations that exist to provide, in general terms, the same representational ability.

A more formal example of a notation, which most everyone is familiar with, is the abstract concept of a counting mechanism or representation; or, to put it in more concrete terms, the concept of a positive integer. The arabic system of enumeration (1, 2, 3, 4, ...) identifies the same abstractions as the roman numerals (I, II, III, IV, ...), and the binary patterns (1, 10, 11, 100, ...). Furthermore, the arithmetic operations on these entities (objects) can be expressed in many different ways (e.g. prefix, infix or suffix notation).

In programming environments we are familiar with the idea of abstract behaviour being represented in different ways. For example, a string of characters can be represented as a fixed array or

as a linked list. Abstract Data Types (ADTs) are useful in computing because they can capture behavioural properties of entities in a manner which allows different implementations to be valid realisations of the same behaviour. In an ADT it is the abstract relationship between entities (objects) which is important. The concrete syntax is inconsequential (except that it should promote understanding of the underlying meaning and be amenable to manipulation within the conceptual framework to which it is being applied).

2.4.2 ADTS in an Object Oriented Semantic Framework

The concrete syntax which we employ in formal object oriented analysis and requirements capture must incorporate the following:

- A means of categorising entities into classes of behaviour.
- A mechanism for representing a set of operations associated with each class, where each operation associates one or more classes of entity with a resulting class of entity. In other words, a means of recording the external interface of a class so that all operations (on class members) can be statically 'type checked' for correctness.
- A means of defining the behaviour associated with each operation. In other words, a set of equations or axioms which give meaning to the operations.
- A facility for defining one class of behaviour in terms of other component classes of behaviour.
- An explicit means of representing the structure of the problem domain.
- Parameterised classes of behaviour (genericity)
- Inclusion polymorphism (subclassing).

Abstract data typing languages provide a suitable formal framework in which these types of property can be expressed. However, the relationship between type and class is complex. ADTs provide a good framework onto which object oriented requirements can be mapped.

An ADT provides us with a means of specifying 'implementation free' behaviour. This is ideal for requirements capture: analysts must try to identify and record *what* is required rather than *how* these requirements are to be met. However, as is argued in the previous section, a set of requirements must always contain some structure otherwise it would be impossible to record or understand them. The object oriented method of analysis and requirements capture encourages the recording of certain structural aspects of the problem domain. This aids understanding and gives the designers an initial structure upon which the design can be developed. In this way a formal statement of object oriented requirements is useful in later stages of development on two accounts: it unambiguously defines what is needed and it provides a structure for understanding the needs.

2.4.3 ADTs in the Initial Stages of Object Oriented Development

The idea of using ADTs at the beginning of an object oriented development strategy is not new. Meyer [84] states that:

> "In object oriented design, every module is organised around a class of data structures ...To avoid implementation dependencies, the underlying description should be that of an ADT."

Meyer goes on to relate design and implementation:

> "Object oriented design is the construction of software systems as structured collections of ADT implementations."

It is clear that Meyer believes that ADTs have an important role to play in the object oriented development of a system. But, Meyer does not further their cause in any way. He gives no indication as to how ADT specifications arise from analysis, or even to show how they can direct the design of an object oriented implementation. Perhaps the use of ADTs is so straightforward that no further instruction is necessary? This thesis shows otherwise.

The relationships between object oriented concepts and parts of an ADT specification are mentioned by Meyer. This informal interpretation is incomplete and imprecise; however, it has enough similarities to the work in this thesis to warrrant inclusion below.

- An ADT expression corresponds to an object.
- The type of the expression corresponds to the class of the object.
- The valid operations on a type correspond to the services (external attributes) which that class of object can provide.
- The algebraic simplification of an expression (as defined in the equations for each operation) can be viewed as equivalent to the internal execution of instructions in an object.
- The value of an object (as an accumulation of its internal state) corresponds to the equivalence group of expressions of which that object is defined to belong in the equations. For example, the equivalence group {'3', '1+2', '2+1', '1+1+1', ...} is represented by the object '3'.

Other object oriented concepts which have not been mentioned in this informal list are primarily structural. The structure of a system which arises from the class hierarchy identified in the analysis is not evident in Meyer's interpretation of ADT specifications. Also, the structure arising from the decomposition of behaviour into component behaviours has been acknowledged only in a very loose way.

2.4.4 A Formal Object Oriented Development Method

It is clear that Meyer's ideas on the role of ADTs in object oriented development need clarification. Before we examine the ADT ACT ONE, and show how it can be used for object oriented requirements capture, it is necessary to re-define the roles of the different stages of development. This acts to put the object oriented requirements capture process in a more concrete context and emphasises the role of Formal Object Oriented Analysis (FOOA). We define the stages of development as follows:

- **Analysis** is the process of understanding a system.

- **Requirements Capture** is the recording of the system understanding as a set of requirements.
- **Design** is the restructuring of the requirements towards an implementation architecture.
- **Implementation** is the realisation of behaviour specified in the design.

This thesis proposes a formal object oriented approach to development. The formal object oriented requirements capture method is defined as follows:

> **Formal object oriented analysis and requirements capture** is the recording of the requirements of a system in terms of a set of ADTs, the structure of which corresponds to the structure of the problem domain. In effect, every object discovered in the analysis has a corresponding ADT specification.

Formal object oriented design is now defined as:

> **Formal object oriented design** is the restructuring of the ADT specifications so that they can be re-used, within a less abstract model, to express the requirements in terms of solution domain objects and architecture.

In object oriented development there is a higher degree of correspondence between problem domain and solution domain structures than with traditional development methods. Designers should be encouraged to re-use analysis structure as much as possible. But, there will always be a conflict of interest between the way in which a problem is recorded and the way it is solved.

2.5 Classes and Types

2.5.1 Typing in Object Oriented Languages: An Introduction

There has been much interest in the relationship between static type checking and dynamic binding in object oriented programming [85, 22, 38]:

- **Static Type Checking:** when the code is statically checked to ensure that all service requests in a system can be fulfilled by the system component receiving objects during the system lifetime.
- **Dynamic Binding:** when the particular methods (code) are bound to service requests at run time.

Object oriented languages which facilitate dynamic binding can give rise to run time errors when objects are asked to provide services which are not part of their interfaces. This is not a desirable feature of any system; in particular critical (real time) systems should not produce 'message-not-known' results. Static type checking can help to prevent such errors. However, as pointed out in [85], such checking can also inhibit a dynamic binding facility. What is required is some means of combining static type checking and dynamic binding which guarantees that no run time errors arise from objects being unable to fulfil requests made of them, whilst also allowing service requests to be dynamically bound to services.

Object oriented programmers have identified the advantages of using abstract data types (ADTs) to support a type checking facility in dynamically bound object oriented languages [38, 22]. This work proposes using ADTs in the analysis and requirements capture stages of object oriented development. Before proceeding to relate the notions of type and class through the concept of data abstraction, it is necessary to examine the notion of type.

2.5.2 Types

A type is a description in the abstract of a related group of entities. Without types it is impossible to reason about all the different objects in a complex system as they would appear to have unrelated behaviour. Typing facilitates the grouping together of values in such a way that the shared behaviour is emphasised whilst the differences are abstracted away from. Typing has three roles:

- **Abstraction:** Values of the same type (in a programming language) share structure and semantics. The structure is used to represent the internal organisation of the value. The semantics represent the external behaviour of the type values. The way in which the values can be interpreted is given by the set of operations applicable to the type and the 'meaning' of such operations.
- **Re-use:** New abstractions can be created from existing ones. Types provide a natural way of structuring libraries into well defined packages of behaviour.
- **Validation:** Types provide a means of guaranteeing the validity of operations on given values through a static analysis of the system in question.

2.5.3 Type Systems

There are many different typing systems which fall into two distinct catagories:

- **Monomorphic** systems require all data values to belong to only one type.
- **Polymorphic** systems allow values to belong to more than one type.

Polymorphism, together with dynamic binding, is a key feature of object oriented systems. In languages where functions are treated as types, the notion of a polymorphic function is widely accepted (see [111, 122], for example). For example, addition is applicable to both integers and reals, and consequently '+' is an **overloaded operation**. It is also possible that '+' can be used to calculate the sum of an integer and a real (with a real result). This is an example of **coercion** — the integer is coerced into being a real value[11]. These polymorphic techniques are available in a wide range of programming languages (imperative, functional and object oriented). They work on only a specific number of types in an unprincipled way. More universal techniques are **genericity** and **subtyping** and these are sometimes refered to as **universal polymorphism** [22].

[11] In object oriented systems, this is similar to a member of one class being dynamically bound to being a member of one of its superclasses.

- **Genericity:** a generic function works universally on a range of types (e.g. a swap function). Unconstrained genericity places no restrictions on the properties exhibited by these types. Constrained genericity is necessary in more complex behaviours. For example, a generic ordered list of values (of the same type) requires the type to have some partial ordering property. Generic types are said to be **parameterised**.
- **Subtyping:** the range of types a function can operate on is determined by a subtyping relationship. A function defined on a type can also operate on any subtypes. In object oriented terms this corresponds to the external attributes of a class including all the attributes of its superclasses. This is also known as **inclusion polymorphism**.

Genericity and subtyping are very different in principle and each have there own place in an object oriented framework [82]. Most abstract data type languages (including ACT ONE) incorporate a facility for defining parameterised types. Parameterised types can be statically instantiated at specification time and as such they give rise to a distinct group of behaviours (which just happen to have a similar structure). Subtyping relationships are much more interesting (in our object oriented framework) because they have an informal correspondence to our notion of subclassing.

2.5.4 Mapping Classes to ADT Specifications

We propose to show that it is beneficial to distinguish the notions of class and type (in the sense of a syntactic interface offered by some element in an implementation language). However, we relate the more formal notion of ADT specification (a well-defined semantic notion of type) with the object oriented concept of class by defining a mapping from object oriented requirements to ADT specification. It is clear that type and class should not be confused [28], but we do believe that types can be used to implement the semantics of the class notion.

Types are more general than classes. In this thesis we generate type specifications from a formal model of object oriented requirements. The set of behaviours that can be specified in this way is much smaller than the set of all behaviours which can be specified using ADTs.

The differences between types and classes (subtypes and subclasses) arise from the way in which the terminology is applied rather than from differences in the underlying principles. The three roles of types, namely abstraction, re-use and validation, are equally applicable to classes:

- **Abstraction:** classes define an abstract interface behind which all the properties of objects in the class are encapsulated.
- **Re-use:** classes provide a fundamental package of re-usable behaviour.
- **Validation:** object oriented systems can be statically analysed to guarantee that all service requests to each object in the system, which may occur in the system lifetime, are available as part of the interface of the class to which the object belongs.

Problems arise in conceptually relating class with type when type is taken to represent a purely static syntactic interface. It is necessary to consider the behaviour offered by type 'members' through their

interfaces. Abstract data types provide both syntactic and semantic views of interface. Consequently, this thesis supports the view that classes and ADTs can be usefully related in a formal framework.

Abstraction is necessary in object oriented analysis since the view of a class as an 'implementation body' is wrong:

- Class defines behaviour — a stack provides LIFO behaviour no matter whether it is implemented in C++ or Eiffel, or whether it is represented (internally) as a linked list or an ordered bag. The notion of class as behaviour is vital when re-use and 'correctness' are considered. When designing a system it is desirable to be able to reason about components of a system without reference to implementation details. This is possible only if the notion of class is implementation independent. Re-use is a behavioural concern: it is wrong to limit re-use to the level of code integration.
- It is not easy to make a distinction between specification and implementation. An abstract specification may have many different valid implementations — an implementation resolves all (or some) of the abstraction. In many cases an implementation of one specification can itself be viewed as a specification of a less abstract set of implementations. There is no clear level of abstraction at which we can distinguish implementation from specification.

We have argued that class is not an implementation concern alone[12]. Implementation classes, as defined in object oriented programming languages, are not to be confused with the notion of class as an abstract statement of behaviour as defined by a particular ADT specification. ADTs provide the foundation upon which object oriented behaviour can be formally modelled. The notions of type and subtype need to be strengthened to provide a formal object oriented interpretation of class and subclass.

ADTs provide an abstraction over data structures in terms of well defined (procedural) interfaces. It is important that classes are not defined solely on the syntax of the interface in their resulting ADT specification. The semantics of behaviour provided at the interface is fundamental in the definition of class relationships. The notion of type as defined by interface is useful only for static type checking in the traditional sense: the non-introduction of syntax errors in code when a type is replaced by a subtype.

Consider the specifications of a queue and a stack. Both specifications could have the same external interface (defined by the operations 'add' and 'remove', say), but it is confusing to say that they have the same type. Type has three roles — abstraction, re-use and correctness. Type as a syntactic interface definition does not fulfil the second role and only partially fulfils the third role. We are not arguing that the notion of typing is without merit. However, within object oriented languages, it is more beneficial to incorporate the type concept in a more powerful means of categorisation, namely classification. This is particularly important when defining inclusion polymorphism. In the following chapters we retain the concept of *type* when referring to purely syntactic properties or relationships.

[12]Classes can be implemented but these implementations define class behaviour in a very constructive fashion — *how* not *what*.

2.6 A Formal Object Oriented Requirements Model in ACT ONE: A Preview

2.6.1 Modelling Object Oriented Requirements in ACT ONE

Type is more general a concept than class. Consequently, we have two options if we wish to use ACT ONE to model object oriented requirements:

- Restrict the ACT ONE syntax (i.e. enforce an object oriented style) and/or incorporate additional static analysis checks to ensure all specifications have a valid object oriented interpretation.
- Define a new object oriented analysis language and provide a mapping from specifications written in this new language to ACT ONE.

We chose the second option because:

- The ACT ONE syntax does not have an object oriented 'flavour'. Although ACT ONE specifications can be given an object oriented interpretation, we feel that it is necessary to have the object oriented concepts prominent in an object oriented statement of requirements.
- ACT ONE is only one particular abstract data typing language. By defining a new language, we have an approach which can be generalised to modelling requirements in any given ADT (or any other formal language).

2.6.2 An Overview of the Class → ADT Mapping

The mapping from object oriented requirements specification to ACT ONE is similar to the mapping suggested by Meyer and others (see section 2.4.3). The mapping transfers the structural and hierarchical aspects of an object oriented model specified in OO ACT ONE to the ACT ONE code. Chapter 3, section 3.5, formalises the mapping from OO ACT ONE to ACT ONE. The fundamental relationships between these two different languages are:

- CLASS → sort.
- LITERALS → literals.
- STRUCTURES → operations which are used to generate the terms which represent members of the class.
- INVARIANTS → global preconditions on sort equations.
- ACCESSOR, DUAL and TRANSFORMER attributes → operations which are term generators.
- Service requests and service responses (i.e. interactions) → evaluation of an ACT ONE expression.
- Composition → parameterisation of structure operations.
- Subclassing → well defined relationship between the classes from which the sorts are generated.
- Inclusion Polymorphism → a form of value coercion between classes and superclasses.

Although the mappings above are only infomally introduced, the flavour of the ACT ONE object oriented model is evident. Chapter 3 adds precision and formality to these informal correspondences.

2.6.3 Using the ACT ONE Object Oriented Model

The ACT ONE generated from OO ACT ONE is used in three ways:

- It helps in the static analysis of object oriented properties in the system being specified.
- It provides an execution model for the testing of dynamic behaviour.
- It provides a natural mode of expression to bridge the gap between analysis and design.

The ACT ONE code is not intended to be explicitly presented to the customer. There are diagrammatic representations of object oriented properties which are more *customer oriented* (see sections 3.2 and 3.3). The structural information in these diagrams corresponds to much of the structural information recorded in the ACT ONE model. The formality underlying the meaning of these object oriented analysis diagrams does not make them any less practical than the widely accepted models advocated in other, less formal, analysis methods.

Chapter 3

An Object Oriented Semantic Framework

3.1 An Overview of the Semantic Framework

The semantic framework, developed in this chapter, connects together the formality and high levels of expressibility of the ADT ACT ONE and our informal understanding of object oriented models, relationships and concepts. At this point in the thesis, our object oriented framework of understanding is dependent on the example object oriented behaviours previously given in chapter 2, together with our own informal conceptualisation of the object oriented paradigm. The object oriented semantic framework is developed to provide a formal model of object oriented concepts which can be used during object oriented analysis and requirements capture. Rather than defining object oriented concepts directly in ACT ONE, a more general approach is proposed in which a new object oriented semantics is developed. This semantics provides a more abstract model which can be implemented by more concrete models. This chapter defines such a semantics and uses the ADT ACT ONE to provide an executable model for the more abstract specifications. The structure of the remainder of this chapter is as follows:

- **Section 3.2: Object-Labelled State Transition System (O-LSTS) Semantics**
  In this section, the semantic model is defined as a particular kind of labelled state transition system (called an O-LSTS[1]). It seems natural to conceptualise the dynamic behaviour of an object as a sequence of states which the object can attain. The state transitions result from the object servicing requests at its external interface. This simple view is expanded upon to encompass the notion of class and relations between classes. A class is defined as a collection of object behaviours which form a set of states which are encapsulated within a common interface. An O-LSTS specification formally defines this notion. A diagramatic representation of an O-LSTS (an O-LSTSD) is introduced as an equivalent way of expressing the information in an O-LSTS specification.

[1] Object-Labelled State Transition System.

- **Section 3.3: An Object Oriented Interpretation of the O-LSTS Model**
  Section 3.3 defines a mapping between the O-LSTS model and the object oriented paradigm. The informal notions of class, object, attribute, service and the relationships between them are given a formal interpretation in our O-LSTS semantics. These definitions add much needed precision to the object oriented terminology. In particular, we formalise two different types of hierarchical model:

  - **Classification model:** the subclassing relationships between all classes in a system are represented in a class hierarchy diagram.
  - **Compositional model:** the structure of an object (in terms of its component parts) is represented in a structure diagram.

  These hierarchical diagrams are used to complement O-LSTSDs.
- **Section 3.4: OO ACT ONE: A Formal Object Oriented Analysis Language**
  This section defines a concrete syntax for the specification of O-LSTSs during analysis and requirments capture. The O-LSTS model is defined in terms of the well understood mathematical notions of sets, cartesian products, relationships and functions. It is necessary to wrap these constructs in a more 'friendly' syntactically sugared syntax. The syntax we define for this purpose is similar to ACT ONE, with a distinctly object oriented flavour: we call it object oriented ACT ONE (OO ACT ONE). Explicit object oriented mechanisms for re-using predefined classes of behaviour (O-LSTSs) are defined. These mechanisms facilitate the definition of different types of subclassing, composition and parameterised classes. Other mechanisms allow the definition of invariant properties, the hiding of internal behaviour and the specification of exceptions. Such a concrete syntax is also necessary when we consider the problem of statically analysing an O-LSTS specification.
- **Section 3.5: An ACT ONE Execution Model for O-LSTS Specifications**
  Section 3.5 provides a mapping from the O-LSTS semantics, as defined in an OO ACT ONE specification, to ACT ONE. This mapping formalises the relationship between object oriented terminology and ADT concepts (for example, type, sort, operation and equation). It should be emphasised that, although ACT ONE specifications can be used to model object oriented requirements, not all ACT ONE specifications have a meaningful object oriented interpretation. The ACT ONE which is produced from OO ACT ONE is used to provide the basis for a static analysis of the typing properties of an OO ACT ONE specification. Furthermore, the ACT ONE provides an 'executable' model for testing the dynamic behaviour of objects defined in OO ACT ONE. The exact nature of this execution model is made clearer in section 3.5.5.

3.2 Object-Labelled State Transition System (O-LSTS) Semantics

The semantic framework is based around the definition of a particular kind of labelled state transition system, namely an Object-LSTS (O-LSTS). It is defined as follows.

3.2.1 Definition: an O-LSTS Specification

An O-LSTS, C_0 say, is a 7-tuple $< O, UTT, HUTT, VTT, HVTT, USS, VSS >$ defined in an environment of O-LSTSs, Env_{C_0}. These eight separate elements of an O-LSTS are formally defined in sections 3.2.1.1 to 3.2.1.8, below.

3.2.1.1 The Environment

Env_{C_0} is specified as a 2-tuple $< C', Rel_{C_0} >$ where,

- C' is a possibly empty set of predefined O-LSTSs $\{C_1, \ldots, C_k\}$, say. We say that C_0 **uses** $C_i, \forall i \in \{1, \ldots, k\}$.
  Definition: Visible Class Set
  The **visible class set** of C_0, written $visible(C_0)$, $= \{C_0\} \cup C' \cup_{i=1}^{k} visible(C_i)$.
- Rel_{C_0} is a set of O-LSTS pairs $\{< C_i, C_j > |\ C_i \sqsubseteq C_j$ and $C_i, C_j \in visible(C_0)\}$.
  Rel_{C_0} represents all explicitly defined subclassing relationships between O-LSTSs visible in C_0. The relationship $\sqsubseteq$ can be defined in many ways[2]. The particular relation which we chose is given in section 3.3.3. Explicit subclassing[3] is reflexive and transitive.
- $\forall < C_i, C_j > \in Rel_{C_k}$, if $C_k \in visible(C_0)$ then $< C_i, C_j > \in Rel_{C_0}$. In other words, a class 'inherits' the subclassing relationships which are defined in the environments of the classes which are visible to it.

3.2.1.2 The Class Members

O is a nonempty set of **typed state labels** $\{O_1, \ldots, O_n\}, n \in \{1, 2, \ldots\}$, called the **typed state set**. Each **typed state label** is either unparameterised, parameterised or conditionally parameterised:

- unparameterised, written as *state-constructor*[4]
- parameterised, written as *state-constructor*$(P_1, \ldots, P_n)$ for $n \in \{1, 2, \ldots\}$ where,
  - i) $\forall P_i \in \{P_1, \ldots, P_n\}$, $P_i \in visible(C_0)$
  - ii) given $S_1(P_1, \ldots, P_r), S_2(Q_1, \ldots, Q_s) \in O$, then $S_1 = S_2 \Rightarrow r = s$ and $P_i = Q_i, \forall i \in \{1, \ldots, r\}$

  Definition: Parameter Classes:
  $P_1, \ldots, P_n$ are called the **parameter classes** of the *state-constructor*.
- conditionally parameterised, written as *state-constructor*$(P_1, \ldots, P_n)$ **on cond**$(P_1, \ldots, P_n)$ for $n \in \{1, 2, \ldots\}$ where,

[2] The O-LSTS model can be said to define a generic formal object oriented framework which is parameterised on the subclassing relationship.

[3] Explicit subclassing relationships are defined with respect to a class environment. We say that $C_i \sqsubseteq C_j$ in Env_{C_0} when C_i is explicitly defined as a subclass of C_j in the environment of C_0.

[4] *State-constructors* are represented as strings of characters — the exact syntax is defined in 3.2.2.1. Conventionally, all other string identifiers, in the O-LSTS definition, are represented in italics.

- i) $\forall P_i \in \{P_1, \ldots, P_n\}$, $P_i \in visible(C_0)$
- ii) given $S_1(P_1, \ldots, P_r), S_2(q_1 : Q_1, \ldots, q_s : Q_s) \in O$, then $S_1 = S_2 \Rightarrow r = s$ and $P_i = Q_i, \forall i \in \{1, \ldots, r\}$
- iii) **cond** is a boolean expression defined on the parameters of the state constructor[5].

Additionally, implicit in every O-LSTS, C_0 say, is an **unspecified state label** which is represented by the **unparameterised** *state-constructor*, written $\sim C_0$.

Definition: *States* function

$States(< O, UTT, HUTT, VTT, VUTT, USS, VSS >) = O$, or $States(C) = O_C$[6].

Definition: Untyped State Set

The **untyped state set** of C_0, written $US(C_0)$, is generated from the **typed state set** as follows:

$US(C_0) = RemoveTypes(States(C_0))$, and

$RemoveTypes(\{o_1, \ldots, o_n\}) = NoType(o_1) \cup \ldots \cup NoType(o_n)$, and

$NoType(state\text{-}constructor) = \{state\text{-}constructor\}$

$NoType(state\text{-}constructor(P_1, \ldots, P_n)) =$
$\{state\text{-}constructor(p_1, \ldots, p_n) \mid p_i \in US(P_i), \forall i \in \{1, \ldots, n\}\}$, and

$NoType(state\text{-}constructor(P_1, \ldots, P_n)$**on cond**$(P_1, \ldots P_n)) =$
$\{state\text{-}constructor(p_1, \ldots, p_n)$ |**cond**$(p_1, \ldots p_n)$ and $p_i \in US(P_i), \forall i \in$
$\{1, \ldots, n\}\}$

The elements of the **untyped state set** are called the **state labels**. Consequently, the **untyped state set** is also known as the **state label set**.

3.2.1.3 External Transformer Interface

UTT is a possibly empty set of **unvalued typed transitions**, called the **unvalued typed transition set**.

$\forall ut \in UTT$, ut is either:

- (i) an **unvalued unparameterised typed transition** of C_0 written as *transition-name*
- (ii) an **unvalued parameterised typed transition** of C_0, written as *transition-name*$< U_1, \ldots, U_r >$, such that $< U_1, \ldots, U_r > \in (visible(C_0))^r$
  The **parameter tuple** of *transition-name*$< U_1, \ldots, U_r >$ is defined to be $< U_1, \ldots, U_r >$.

Given an **unvalued typed transition set**, it is necessary to generate the set of all unvalued transitions through an actualisation of all possible combinations of **parameter tuple** values. The set generated is defined as follows.

Definition: Unvalued Actualised Transition Set

[5] The syntax and semantics of boolean expressions is defined by a **state label expression** with **type** boolean — see 3.2.3.

[6] Similarly, $UTT_C, HUTT_C, VTT_C, HVTT_C, USS_C, VSS_C$ represent the 2nd, 3rd, 4th, 5th, 6th and 7th elements of the 7-tuple O-LSTS C.

$UAT(UTT) = RemoveUParameters(UTT)$ where,

$RemoveUParameters(\{\}) = \{\}$, and
$RemoveUParameters(\{ut_1, \ldots, ut_n\})$
$= ActUParameters(ut_1) \cup \ldots \cup ActUParameters(ut_n)$, where

$ActUParameters(\textit{transition-name}) = \{\textit{transition-name}\}$
$ActUParameters(\textit{transition-name} < U_1, \ldots, U_n >) =$
$\{\ \textit{transition-name}(u_1, \ldots, u_n) \mid u_i \in U_i, \forall i \in \{1, \ldots, n\}\}.$

3.2.1.4 Hidden Transformers

$HUTT$ is a subset of UTT called the **hidden unvalued typed transition set**. We define an **unhidden unvalued typed transition** to be any member of UTT which is not a member of $HUTT$.

3.2.1.5 External Accesors Interface

VTT is a possibly empty set of valued typed transitions, called the **valued typed transition set**. $\forall vt \in VTT$ vt is either:

- (i) a **valued unparameterised typed transition** of C_0, written as *transition-name*:V_{vt}, where: $V_{vt} \in visible(C_0)$ is called the **result type** of the transition
- (ii) a **valued parameterised typed transition** of C_0, written as *transition name*$< U_1, \ldots, U_r >$: V_{vt}, such that $V_{vt} \in visible(C_0)$ and $< U_1, \ldots, U_r > \in (visible(C_0))^r$
  The **parameter tuple** of *transition-name*$< U_1, \ldots, U_r >$:V_{vt} is defined to be $< U_1, \ldots, U_r >$.

Given a **valued typed transition set**, it is necessary to generate the set of all valued transitions through an actualisation of **parameter tuple** values. The set generated is defined as follows.

Definition: Valued Actualised Transition Set

$VAT(VTT) = RemoveVParameters(VTT)$, where

$RemoveVParameters(\{\}) = \{\}$ and
$RemoveVParameters(\{ut_1, \ldots, ut_n\})$, for $n \in \{1, 2, \ldots\}$,
$= ActVParameters(ut_1) \cup \ldots \cup ActVParameters(ut_n)$, where

$ActVParameters(\textit{transition-name}{:}\mathrm{V}) = \{\textit{transition-name}\}$
$ActVParameters(\textit{transition-name} < U_1, \ldots, U_n > : V) =$
$\{\ \textit{transition-name}(u_1, \ldots, u_n) \mid u_i \in U_i, \forall i \in \{1, \ldots, n\}\}.$

The **result type** of a **valued actualised transition** is defined to be the **result type** of the **valued typed transition** from which it was generated.

3.2.1.6 Hidden Accessors

$HVTT$ is a subset of VTT, called the **hidden valued typed transition set**. We define an **unhidden valued typed transition** to be any member of VTT which is not a member of $HVTT$.

3.2.1.7 Transformer Behaviour

USS is a possibly empty set of **unvalued state-to-state transitions** $\{From_{O_j} : O_j \in US(C_0)\}$, one, and only one, for every $O_j \in US(C_0)$, where $From_{O_j} \subseteq UAT(UTT) \times US(C_0)$.
Now, $\forall O_j \in US(C_0)$ the following **completeness conditions** must be upheld:

- (i) given $ul \in UAT(UTT)$, $\exists O_k \in US(C_0)$ such that $< ul, O_k > \in From_{O_j}$
- (ii) given $< ul1, O_1 > \in From_{O_j}$ and $< ul2, O_2 > \in From_{O_j}$, $ul1 = ul2 \Rightarrow O_1 = O_2$

The **unvalued state-to-state transitions** from the unspecified state $\sim C_0$ do not have to be explicitly defined. Unless otherwise specified, $< ul, \sim C_0 > \in From_{\sim C_0}, \forall ul \in UAT(UTT)$.

3.2.1.8 Accessor Behaviour

VSS is a possibly empty set of **valued state-to-state transitions** $\{Valfrom_{O_j} : O_j \in O\}$, one, and only one, for every $O_j \in US(C_0)$, where $Valfrom_{O_j} \subseteq VAT(VTT) \times visible(C_0) \times US(C_0)$
Now, $\forall O_j \in US(C_0)$ the following **completeness conditions** must be upheld:

- (i) given $vl \in VAT(VTT)$ with **result type** V, $\exists O_k \in US(C_0)$ and $res \in US(V)$ such that $< vl, res, O_k > \in Valfrom_{O_j}$
- (iii) given $< vl1, res1, O_1 >, < vl2, res2, O_2 > \in Valfrom_{O_j}$, $vl1 = vl2 \Rightarrow res1 = res2$ and $O_1 = O_2$

The **valued state-to-state transitions** from the unspecified state $\sim C_0$ do not have to be explicitly defined. Unless otherwise specified, $< vl, \sim C', \sim C_0 > \in ValFrom_{\sim C_0}, \forall vl \in VAT(VTT)$, where C' is the **result type** of the **valued typed transition** vl.

3.2.1.9 Additional Syntactic Constraints

The following additional syntactic constraints are defined to enable **state labels** and **typed transitions** to be uniquely identified. They also make the O-LSTS models easier to translate to ACT ONE (see section 3.5). The additional constraints are as follows:

- *State-Constructors* must be uniquely defined as being unparameterised, parameterised or conditionally parameterised.
- All *state-constructors* and *transition-names* are uniquely defined in each O-LSTS by strings of characters. These strings can include alphanumeric characters (and the '-' character for constructing structured strings). The final character must not be a digit.
- *Transition-names* must not correspond to *state-constructors*.
- **State labels** in the USS and VSS tuples must be represented as **state label expressions** (see 3.2.3).

3.2.2 O-LSTS Examples

To clarify the formal definition, some examples follow. These examples do not illustrate every aspect of the O-LSTS model. In particular they do not show the significance of the subclass hierarchy defined in the environment of an O-LSTS. The examples specify purely object based systems. We examine the specification of object oriented systems after defining subclassing in 3.3.3.

O-LSTS Example 1: Resetable-Traffic-Light

The environment of the O-LSTS Resetable-Traffic-Light is defined to be $< C', \{\} >$, where $C' = \{$ Bool, traffic-light-colour$\}$, $States$(Bool) $= \{true, false\}$ and $States$(traffic-light-colour) $= \{red, green, amber\}$ Resetable-Traffic-Light $= < O, UTT, HUTT, VTT, HVTT, USS, VSS >$, where

- $O = \{t-l(traffic-light-colour)\}$
  (Consequently, $US(Resetable-Traffic-Light) = \{t-l(red), t-l(green), t-l(amber)\}$)
- $UTT = \{next, reset < traffic-light-colour >\}$
  Consequently, $UAT(UTT) = \{next, reset(red), reset(green), reset(amber)\}$
- $HUTT = \{\}$
- $VTT = \{safe-to-go$:Bool, eq <Resetable-Traffic-Light>: Bool$\}$
  Consequently, $VAT(VTT) = \{safe{-}to{-}go$:Bool, $eq(t{-}l(red))$:Bool, $eq(t{-}l(amber))$:Bool, $eq(t{-}l(green))$:Bool$\}$.
- $HVTT = \{\}$
- $USS = \{From_{O_j} : O_j \in O\}$ where
  - $From_{t-l(red)} = \{< next, t-l(green) >, < reset(amber), t-l(amber) >, < reset(green), t-l(green) >, < reset(red), t-l(red) >\}$
  - $From_{t-l(amber)} = \{< next, t-l(red) >, < reset(amber), t-l(amber) >, < reset(green), t-l(green) >, < reset(red), t-l(red) >\}$
  - $From_{t-l(green)} = \{< next, t-l(amber) >, < reset(amber), t-l(amber) >, < reset(green), t-l(green) >, < reset(red), t-l(red) >\}$
- $VSS = \{ValFrom_{O_j} : O_j\}$ where
  - $ValFrom_{t-l(green)} = \{< safe-to-go, true, t-l(green) >, < eq(t-l(red)), false, t-l(green) >, < eq(t-l(amber)), false, t-l(green) >, < eq(t-l(green))true, t-l(green) >\}$
  - $ValFrom_{t-l(red)} = \{< safe-to-go, false, t-l(red) >, < eq(t-l(green)), false, t-l(red) >, < eq(t-l(amber)), false, t-l(red) >, < eq(t-l(red)), true, t-l(red) >\}$
  - $ValFrom_{t-l(amber)} = \{< safe-to-go, true, t-l(amber) >, < eq(t-l(green)), false, t-l(amber) >, < eq(t-l(amber)), true, t-l(amber) >, < eq(t-l(red)), false, t-l(amber) >\}$

This 7-tuple is a valid O-LSTS, since it fulfils all the necessary and sufficient conditions of the definition. It is represented by the O-LSTS Diagram (O-LSTSD) in figure 3.1.

The O-LSTSD is a rectangle containing a graph of labelled nodes and links. The class name is given above the rectangle and the environment of the class is defined by the list of O-LSTSs following the `using` keyword and `NO HIERARCHY` specifies that there are no explicit class relationships to consider. Each node in the diagram contains a unique **state label**. All **state labels** in the O-LSTS are represented by nodes. The set of transitions between nodes (represented by the set of links) is isomorphic to the union of the **state-to-state transition sets**. In other words, $\forall < ua, O_k > \in From_{O_j}$, $\exists$ a unique *Node-LabelledLink-Node* connection, in the O-LSTSD, from O_j to O_k. Similarly, $\forall < va, val, O_l > \in Valfrom_{O_m}$, $\exists$ a unique *Node-LabelledLink-Node* connection, in the O-LSTSD,

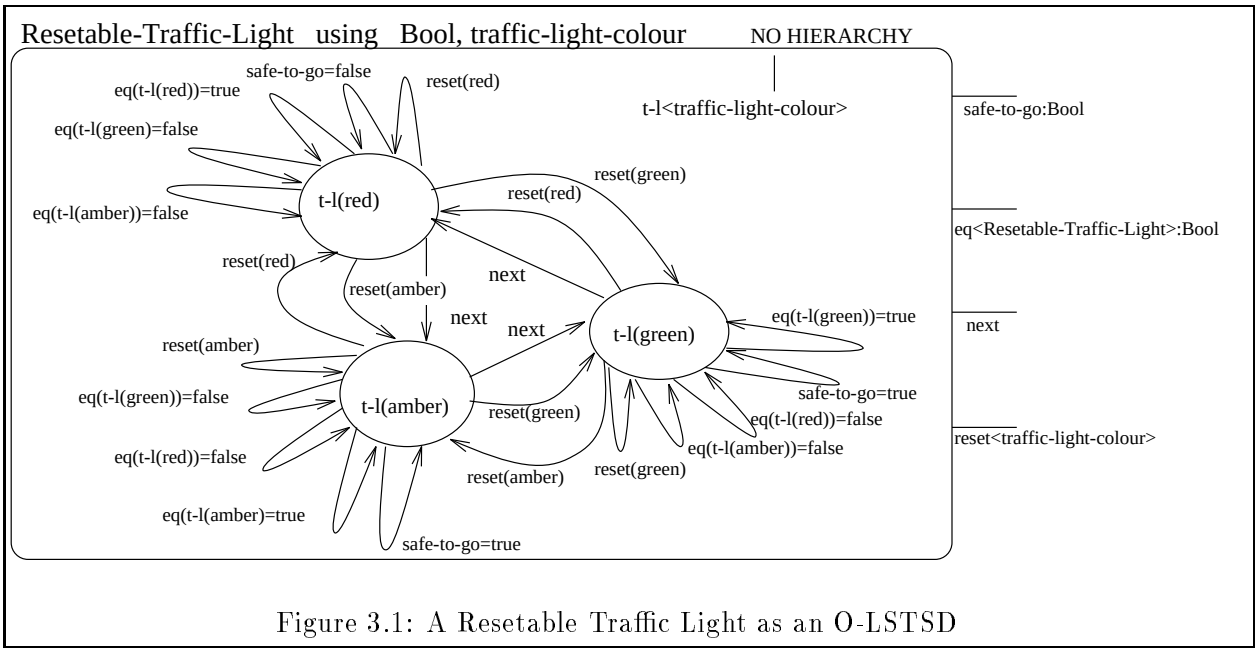
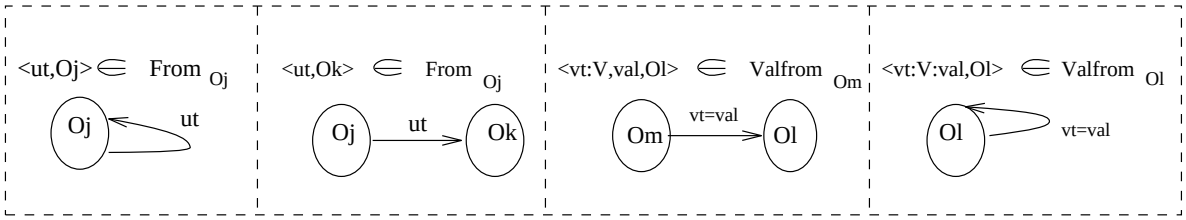


Figure 3.1: A Resetable Traffic Light as an O-LSTSD

from O_l to O_m. (When $O_j = O_k$, or $O_l = O_m$, the link connects the node with itself.) These four cases are illustrated in the diagram below:



The (valued and unvalued) **typed transition sets** are represented by connections to the outside of the rectangle around the O-LSTSD. The **hidden** transitions (there are none in this system) must be identified by appending the transition name with the label `HIDDEN`. Similarly, the **state label types** are represented by connections on the inside of the rectangle, together with the conditions placed on the parameters (if there are any).

Diagramatic Syntactic Sugaring

As even simple O-LSTSDs get very cluttered with nodes and links, there are a number of extensions which can be used to sugar the diagramatic representation. In the example above, an obvious extension is to parameterise the result of the reset transition from each node. This is done in figure 3.2. The dotted link represents a set of transitions (one for each value the parameter can take, i.e. one for every member of the **state label set** of the **parameter type**). The result of a parameterised transition is a parameterised node (a node with a dotted circumference whose **state label** is also parameterised). All parameters in an O-LSTSD are represented between diamond brackets.

It can be seen from figure 3.2 that the parameterised resets are the same for each node. A further sugaring permits the parameterisation of the node labels at both ends of a transition. The result

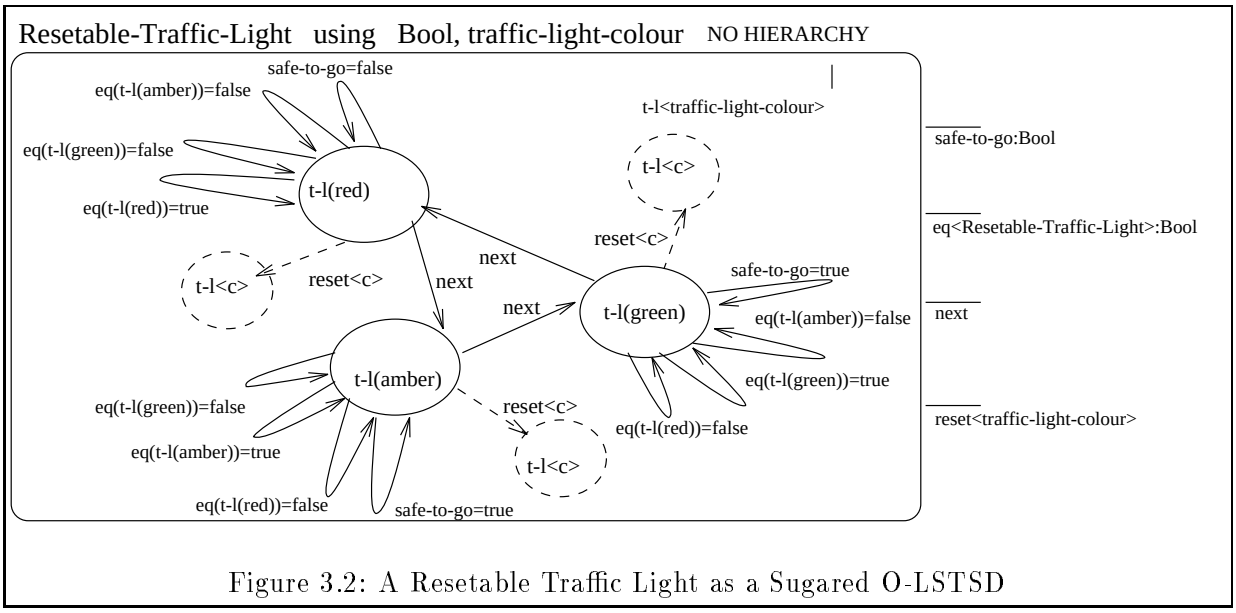


Figure 3.2: A Resetable Traffic Light as a Sugared O-LSTSD

of a transition is a parameterised expression (also in diamond brackets), [7] which is dependent on the transition parameter values and the **state label** parameter values of the node from which the transition is taking place. Two examples of such a parameterisation are illustrated in figure 3.3.

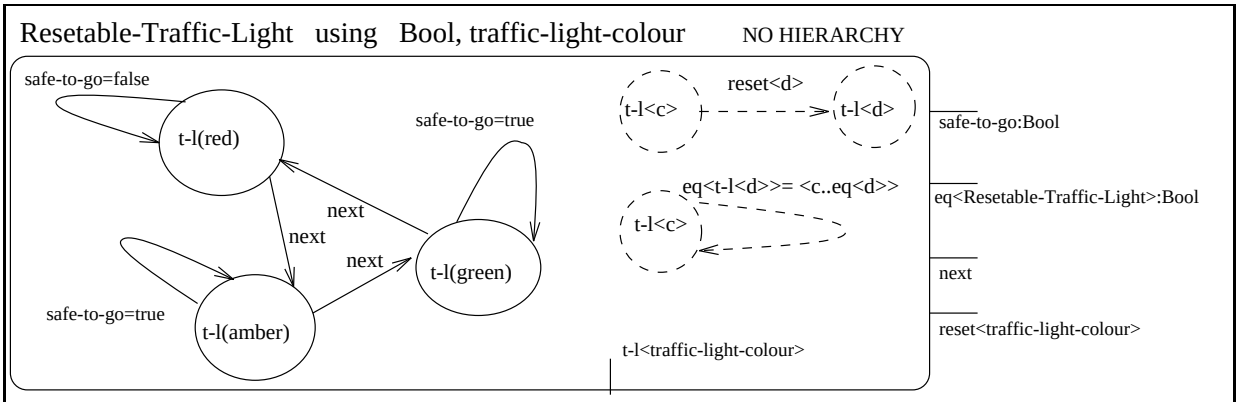


Figure 3.3: A Further Sugaring of the O-LSTSD

O-LSTS Example 2: An Integer Counter

The resetable traffic light example illustrates the specification of a system with a finite number of states in which it is not necessary to parameterise the behaviour definition, although the parameterisation does simplify and clarify the specification. It is necessary to provide facility for defining O-LSTSs in a parameterised fashion. An unbounded integer counter, for example, cannot be represented by a finite state machine and so we must provide a suitable means of defining an infinite set of behaviours

[7] The syntax and semantics of such **state label expressions** is formalised later in this section. For now, the interpretation is informal, but intuitive and well explained by the examples.

in a parameterised form. Consider the O-LSTSD in figure 3.4.

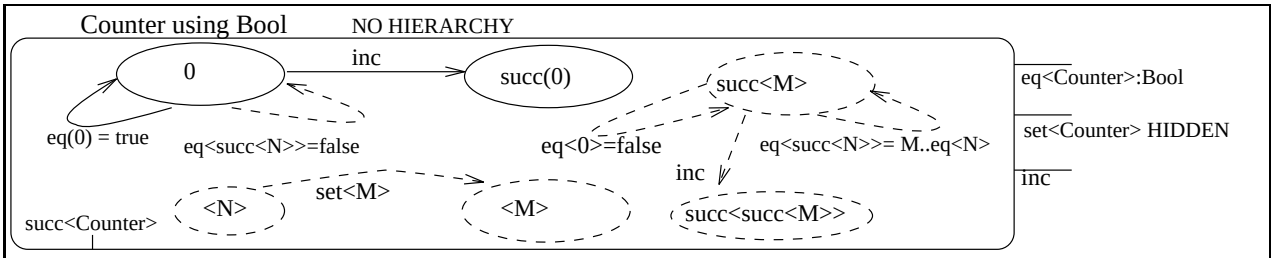


Figure 3.4: An O-LSTSD Specification of an Integer Counter

The following should be noted:

- The connection on the inside of the diagram is labelled by the expression $succ < Counter >$. This specifies that $\forall n \in US(Counter), succ(n) \in US(Counter)$.
- The inc transition from 0 to $succ(0)$ specifies that $< inc, succ(0) > \in From_0$.
- The $eq(0) = true$ transition from 0 to 0 specifies that $< eq(0), true, 0 > \in Valfrom_0$.
- The parameterised transition $eq(succ < N >) = false$ from 0 to 0 specifies that $\forall N \in US(Counter)$,
  $< eq(succ(N)), false, 0) \in ValFrom_0$.
- The parameterised transition $eq(0)$ from $succ < M >$ back to itself specifies that $\forall M \in US(Counter)$,
  $< eq(0), false, succ(M) > \in ValFrom_{succ(M)}$.
- The parameterised transition inc from $succ < M >$ to $succ < succ < M >>$ specifies that $\forall M \in US(Counter)$,
  $< inc, succ(succ(M)) > \in From_{succ(M)}$.
- The parameterised transition $eq < succ < N >>= M..eq(N)$ from $succ < M >$ back to itself specifies that $\forall N, M \in US(Counter), < eq(succ(N)), M..eq(N), succ(M) > \in From_{succ(M)}$. (In this case $M..eq(N)$ is a **state label expression** which represents a **state label** in the **untyped state set** of Bool The meaning of such a **state label expression** is defined in the next section (3.2.3)).
- The parameterised transition $set < M >$ from N to M specifies that $\forall N, M \in US(Counter)$, $< set < M >, M > \in From_N$.

The O-LSTSD is equivalent to the OLSTS specification:
$Counter = < O, UTT, HUTT, VTT, HVTT, USS, VSS >$ in $< \{Bool\}, \{\} >$, where

- $O = \{0, succ(Counter)\}$,
  consequently $US(Counter) = \{0, succ(0), succ(succ(0)), \ldots\}$
- $UTT = \{inc, set < Counter >\}$,
  consequently $UAT(UTT) = \{inc\} \cup \{set(n) \mid n \in O\} = \{inc, set(0), set(succ(0)), \ldots\}$
- $HUTT = \{set < Counter >\}$. This is the first example of a **hidden** transition.

- $VTT = \{eq < Counter >: Bool\}$,
  consequently $VAT(VTT) = \{eq(n) \mid n \in O\} = \{eq(0), eq(succ(0)), eq(succ(succ(0)), \ldots\}$
- $HVTT = \{\}$.
- $USS = \{From_x \mid x \in O\}$, where
  $\forall x \in O$, $From_x = \{< inc, succ(x) >\} \cup \{< set(n), n > \mid n \in O\}$
- $VSS = \{ValFrom_y \mid y \in O\}$, where
  $ValFrom_0 = \{< eq(0), true, 0 >\} \cup \{< eq(succ(n)), false, 0 > \mid n \in O\}$, and
  $\forall y \in O$, $ValFrom_{succ(y)} = \{< eq(0), false, succ(y) >\} \cup \{< eq(succ(p), q.eq(y), succ(y) > \mid p \in O\}$.

3.2.3 State Label Expressions

The **state labels** which represent the newstate of an object after it services a request and the value returned by a valued attribute can be defined by an expression (called a **state label expression**) which evaluates to a **state label**. The syntax and semantics of such expressions are defined below.

C is the **serving class**, sl is the **server** and att is the **service** of the **state label expression**. When a **state label expression** is such that the **service class** cannot be uniquely identified from the **server** and the **serving class**, then the class identifier must be included in the expression to remove the risk of ambiguity. For example, in a system which uses the Nat O-LSTS, $M.eq(N)$ must be represented by $M{:}Nat.eq(N)$ if Nat is not the only visible class of the system which has $eq < Nat >$ as a **typed transition** and M as a **state label**.

Additional Hiding Constraint

The members of $HUTT$ and $HVTT$ model internal transitions of a class which are available only to the class in which they are defined. Consequently, we place the additional syntactic constraint on **state label expressions** that if $C_0 \neq C$ then att, the **service**, must be an **unhidden transition** of C.

3.3 An Object Oriented Interpretation of the O-LSTS Model

Section 3.3 identifies five relationships which collectively must exist in an analysis model for it to be considered object oriented: classification, interaction, subclassing, composition and configuration. Classification and subclassing are static properties of an object oriented system. Composition and configuration have both static and dynamic aspects, although during analysis they are most usefully given a static interpretation. Interaction is the only purely dynamic property of an object oriented system which is important in analysis.

Definition: state label expression
A **state label expression** in an O-LSTS (C_0) definition is said to be defined on a class C, where $C \in visible(C_0)$. Such an expression is either:

- i) a simple expression, written sl, where $sl \in US(C)$.
  C is called the **type** of the simple expresion.
- ii) a transition expression, written $sl.att$, where
  - $sl \in US(C)$
  - att is an unparameterised transition, or
    att is a parameterised transition $att(p_1, \ldots, p_n)$ such that $\forall i \in \{1, \ldots, n\}$:
    $p_i \in US(P_i)$ and $att(Q_1, \ldots, Q_n) \in UTT_C$ and $P_i \sqsubseteq Q_i$ in Env_{C_0}

  We define $sl.att$ to be an equivalent representation of the **state label** sl', where either $< att, sl' > \in From_{sl} \in USS_C$ or sl'' where $< att, val, sl'' > \in ValFrom_{sl} \in VSS_C$).
  C is called the **type** of a transition expression.
- iii) a result expression, written $sl..att$, where
  - $sl \in US(C)$, and
  - att is an unparameterised value transition, or
    att is a parameterised value transition $att(p_1, \ldots, p_n)$ such that $\forall i \in \{1, \ldots, n\}$: $p_i \in US(P_i)$ and $att(Q_1, \ldots, Q_n) \in VTT_C$ and $P_i \sqsubseteq Q_i$ in Env_{C_0}

  We define $sl..att$ to be val, where $< att, val, sl' > \in From_{sl} \in USS_C$.
  The **type** of a result expression is the **result type** of the **valued actualised transition attribute** att.

3.3.1 O-LSTS Classification

An interpretation of classification in the O-LSTS model is given through the following definitions of class, object and external interface[8].

- **Class definition**
  An O-LSTS defines a class of behaviour. A class is specified by:
  - The set of class members, which is defined by the **typed state label set** in the O-LSTS definition.
  - The external interface (a set of attributes), which corresponds to the **unhidden typed transitions** $(UTT \setminus HUTT) \cup (VTT \setminus HVTT)$ in the O-LSTS definition.
  - The behaviour of the member objects in response to service requests at their external interface, which corresponds to the behaviour defined by the **state-to-state transition sets** of the O-LSTS definition ($USS \cup VSS$).

[8]The definitions of class, object and external interface are mutually dependent — like the chicken and the egg, it is difficult to say in which order they come!

- **Object definition**
  The term object is used to represent a dynamic instance of a member of a class. In the definition of a class as an O-LSTS, the set of **state labels** are used only to represent the set of possible values (or states) that an object of the class can attain. An object of a class must be associated with a unique identification, which is then used to provide a reference to a particular member of the **state label set** of the class (O-LSTS) of which the object is a member. Consequently, the *state* of an object is defined precisely by the **typed state label** being referenced by the object. The external interface of an object is defined by the external interface of the class to which it belongs. The behaviour provided by the external interface of an object is defined in the **valued** and **unvalued transition sets** of its class ($From_{object}$ and $ValFrom_{object}$).
- **External Interface definition**
  The external interface of a class is defined as a set of attributes. These external attributes are categorised as follows:

  - **Dual:**
    a service request at a dual attribute results in the receiving object updating its internal state and returning some result value to the service requester.
  - **Transformer:**
    a service request at a transformer attribute results in the receiving object updating its internal state without returning any result value to the requester.
  - **Accessor:**
    a service request at an accessor attribute results in the receiving object returning a result value to the requester with no change to its state. (An accessor is a particular type of dual in which the state is 'updated' to the value it was previously.)

  This categorisation is reflected in the O-LSTS semantics as follows:

  - **Dual attributes** are defined by the **valued state-to-state transitions** of an O-LSTS. For example, a **dual attribute** of an object O_j is represented by: al, say, such that $< al, result, O_k > \in Valfrom_{O_j}$
  - **Transformer attributes** are defined by **unvalued state transitions** of an O-LSTS. For example, a **transformer attribute** of an object O_j is represented by: al, say, such that $< al, O_k > \in From_{O_j}$
  - **Accessor attributes** are particular types of **dual attributes** in which the state of the object is not affected by fulfilling the accessor service. In other words, an accessor attribute of an object O_j is represented by: al, say, such that $< al, result, O_j > \in Valfrom_{O_j}$

3.3.2 O-LSTS Interaction: The Executable Semantics

Dynamic behaviour of an object oriented system corresponds to the behaviour of the object representing the system. Object behaviour is defined as a sequence of interactions between the object,

O say, and its external environment[9]. The environment of O is made up of a set of **service requesters** (themselves objects) which interact with O, the **service provider**, by requesting services of its external interface.

Creating an object of a class, C say, corresponds to referencing the unique identification for the new object with one of the **state labels**, O_j say, in the set $States(C)$. When the object receives a service request which corresponds to an external attribute, the object behaves as follows:

- When the corresponding external attribute is a transformer, $ua \in UTT$ say, then, by definition, there exists one and only one $< ua, O_k > \in From_{O_j}$. The new state referenced by the object is O_k, and this can be represented, without risk of ambiguity, by the **state label expression** $O_j.ua$. The object does not return any value to the service requester and it proceeds to fulfil the external behaviour as defined for O_k in the O-LSTS C.
- When the corresponding external attribute is a dual, $va : V \in VTT$ say, then, by definition, there exists one and only one $< va : V, result, O_k > \in ValFrom_{O_j}$. The new state of the object is set to be O_k (which can be represented by the **state label expression** $O_j.va$) and the value returned to the service requester is a reference to the **state label** $result$ in the O-LSTS V (which can be represented by the **state label expression** $O_j..va$). The object then behaves like O_k in C.

An implementation of this dynamic model is defined by a mapping from O-LSTS specifications to ACT ONE. The evaluation of certain ACT ONE expressions corresponds to the processing that an object performs in response to a service request (see 3.5 for more details).

3.3.3 O-LSTS Subclassing (and Subtyping)

A subclassing relationship between classes is defined as a relationship between the O-LSTSs corresponding to these classes. Informally, there are four constraints which must be fulfilled for one O-LSTS, A say, to be a subclass of another O-LSTS, B say, (written $A \sqsubseteq B$):

- i) A must provide the external interface of B. If B can service a particular request (i.e. if all the members of B can service the request) then A must also be able to service that request. This is a subtyping relationship.
- ii) All members of A must also be members of B. More precisely, all ways of identifying member objects of class A must also be valid identifications for members of class B.
- iii) The members of A must offer the same behaviour as their corresponding members in B. In other words, it must be impossible to distinguish between corresponding members of A and B by requesting services of these objects which B is capable of fulfilling (we abstract away from the fact that A may offer services which B does not offer).

[9]Note that the environment of an object is not directly related to the environment of the class to which it belongs. The environment of a class represents a set of classes which are used in its definition.

- iv) The environment of classes which B uses must be 'contained' (in some way) in the environment of classes which A uses. The exact definition of containment must take into account the relative visibility of classes and the subclassing relationships between them.

These (informal) necessary and sufficient conditions are formally specified in section 3.3.3.3. The first two conditions are purely syntactic constraints which are based on the static properties of the classes concerned. The third condition is a semantic constraint founded on the dynamic behaviour of the classes. The fourth condition needs examination only when the environments of the two classes are different.

Informally, subclassing is similar to the mathematical notion of the subset relationship. Before proceeding to define subclassing, we review our claim that subclassing is not the same as subtyping. To do this, we formally define a **subtyping** relationship between O-LSTSs and argue that **subtyping** is a necessary, but not sufficient, condition for subclassing (in the intuitive sense). Then we formally define **subclassing** between O-LSTSs and prove that **subclassing** $\Rightarrow$ **subtyping** and **subtyping** $\not\Rightarrow$ **subclassing**

3.3.3.1 Subtyping

Subtyping between O-LSTSs guarantees that any object of a given class, C say, can be replaced by an object which is a member of any subtype (class) of C without introducing the possibility of *syntax errors*[10] into the system in which the replacement is made. Subtyping between O-LSTSs must similarly guarantee only the non-introduction of *syntax errors* when a subtype is used to provide a

[10] In object oriented systems, a *syntax error* results when an object cannot respond to a service requested of it.

replacement for a supertype. Such a relationship is defined below.

Definition: O-LSTS Subtyping ($\leq$)
$< O, UTT, HUTT, VTT, HVTT, USS, VSS >$in $Env_1 \leq$
$< O', UTT', HUTT', VTT', HVTT', USS', VSS' >$in $Env_2 \Leftrightarrow$

- $(UTT \setminus HUTT) = (UTT' \setminus HUTT')$ or
  - (i) given an **unvalued unhidden unparameterised typed transition** of UTT', written *transition-name*, then *transition-name* $\in (UTT \setminus HUTT)$.
  - (ii) given an **unvalued unhidden parameterised typed transition** of UTT', written $t\text{-}n< U'_1, \ldots, U'_r >$, $\exists$ $t\text{-}n< U_1, \ldots, U_r > \in (UTT \setminus HUTT)$, such that $U'_i \leq U_i, \forall i \in \{1, \ldots, r\}$.
- $VTT = VTT'$ or
  - (iii) given a **valued unhidden unparameterised typed transition** of VTT', written *transition-name*:V',
    $\exists$ *transition-name*:$V \in (VTT \setminus HVTT)$, such that $V \leq V'$.
  - (iv) given a **valued unhidden parameterised typed transition** of UTT', written $t\text{-}n< U'_1, \ldots, U'_r >: V'$,
    $\exists$ $t\text{-}n< U_1, \ldots, U_r >: V \in (VTT \setminus HVTT)$ such that:
    a) $U'_i \leq U_i, \forall i \in \{1, \ldots, r\}$ and
    b) $V \leq V'$

Conditions (ii) and (iv a) correspond to the 'rule of contravariance' for subtyping — a subtype can accept parameter values which are supertypes of the values which the supertype can accept. Conditions (iii) and (iv b) correspond to the 'rule of covariance' for subtyping — a subtype can respond with values which are subtypes of the values which the supertype responds with. **Subtyping** is reflexive and transitive. Note that the **hidden** transitions are not important in the subtyping relation. When two O-LSTS are subtypes of each other they are said to be **type compatible**:

Definition: Type Compatibility
Two O-LSTSs, A and B, are **type compatible**, written $A \leq\geq B$, $\Leftrightarrow$ A $\leq$ B and B $\leq$ A.

Type compatibility is an equivalence relation.

Example 1: A Simple Subtyping Relationship

The *Store-evens* and *Store-fours* classes in figure 3.5 illustrate a nontrivial **subtyping** relationship. The environments of these O-LSTSs are composed from the O-LSTSs *evens*, *fours* and *eights*, which are defined such that $evens \leq fours \leq eights$, and $States(evens) = \{2, 4, 6, 8\}$, $States(fours) = \{4, 8\}$ and $States(eights) = \{8\}$. Consequently, by the **subtyping** definition, *Stores-fours* $\leq$ *Stores-evens*. The rule of contravariance is upheld since $evens \leq fours$. The rule of covariance is upheld since $fours \leq eights$.

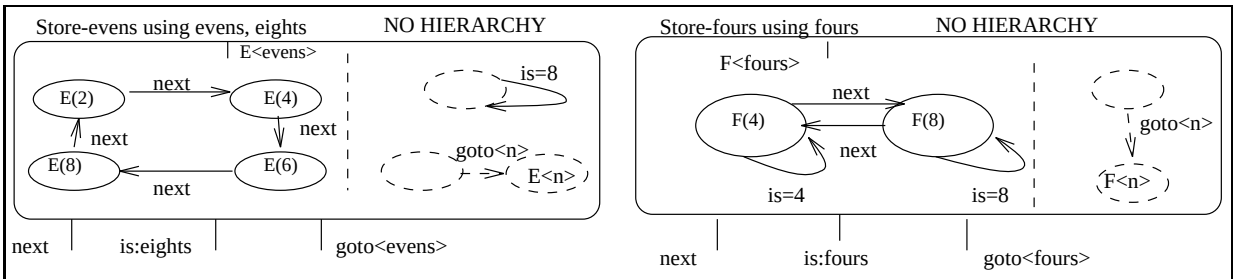


Figure 3.5: Subtyping: A Simple Example

3.3.3.2 Additional Syntactic Constraint Between State Label Sets

From the previous examples, it is clear that **subtyping** does not place any requirements on the representation of **state labels**: it is purely a relationship between O-LSTS interfaces. In our informal definition of subclassing it was stated that all members of a class must also be members of its superclass(es). As O-LSTS members are **state labels** (or more precisely references to **state labels**) of the O-LSTS, it is necessary to place the additional restriction that the **state label** sets of O-LSTSs, which are related by the subclassing relationship, must be related by a subset relationship. This additional syntactic constraint is a necessary but not sufficient condition for **subclassing**. Example 2 shows that when both syntactic conditions hold a subclassing relationship is still not guaranteed.

Example 2: Subtyping is not Subclassing

At the beginning of this section, **subtyping** was said to be 'too weak' to be equated to our informal behavioural notion of subclassing. Subclassing between classes of behaviour requires a formal relationship between the behaviour offered by the members of each of the classes. The need for this additional behavioural requirement is re-iterated by the O-LSTSs defined in figure 3.6.

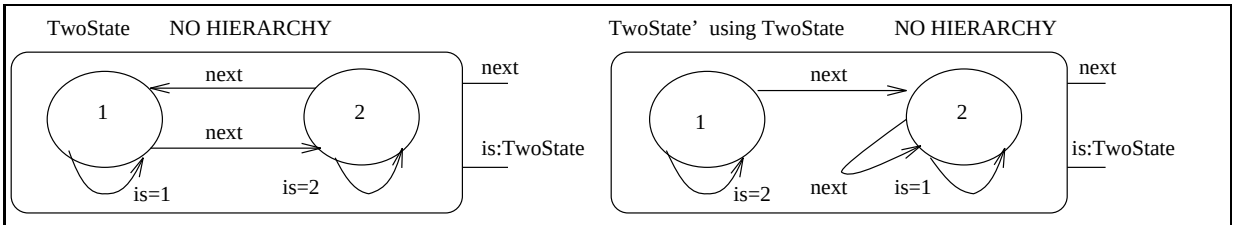


Figure 3.6: Subtyping is not Subclassing: An Example

The two O-LSTSs are **type compatible**, even though the behaviour offered by them is quite different. It is not possible to replace a member of one O-LSTS with a member of the other whilst guaranteeing the behaviour of the system in which the change is made. In this case the **subtyping** relationship guarantees only the non-introduction of *syntax errors* when a member of one class is replaced by the corresponding member of another. Consequently, a semantic relationship (in which replacement somehow guarantees behavioural compatibility) must address the relationship between the **state-to-state transition sets** of O-LSTSs, and not just their external interfaces. This semantic

requirement is precisely the notion we capture in our **subclassing** relationship. We return to the formalisation of this relationship in the knowledge that **subtyping** must be a necessary but not sufficient condition for **subclassing**.

3.3.3.3 Subclassing

Definition: Subclassing ($\sqsubseteq$)
Given A, specified as $< O, UTT, HUTT, VTT, HVTT, USS, VSS >$ and B,
specified as $< O', UTT', HUTT', VTT', HVTT', USS', VSS' >$ then $A \sqsubseteq B \Leftrightarrow$

- i) $US(A) \subseteq US(B)$.
- ii) Every **unvalued unhidden unparameterised typed transition** of B is also an **unvalued unhidden unparameterised typed transition** of A.
- iii) For every **unvalued unhidden parameterised typed transition** of B, written *transition-name*$< U_1, \ldots, U_r >$, there is an **unvalued unhidden parameterised typed transition** of A, written *transition-name*$< V_1, \ldots, V_r >$, such that $U_i \sqsubseteq V_i$ in $Env_A, \forall i \in \{1, \ldots r\}$.
- iv) For every **valued unhidden unparameterised typed transition** of B, written *transition-name*$:Val_B$, there is a **valued unhidden unparameterised typed transition** of A, written *transition-name*$:Val_A$, such that $Val_A \sqsubseteq Val_B$ in Env_A.
- v) For every **valued unhidden parameterised typed transition** of B, written *transition-name*$< U_1, \ldots, U_r >: Val_B$, there is an **valued unhidden parameterised typed transition** of A, written *transition-name*$< V_1, \ldots, V_r >: Val_A$, such that $U_i \sqsubseteq V_i$ in $Env_A, \forall i \in \{1, \ldots r\}$ and $Val_A \sqsubseteq Val_B$ in Env_A.
- vi) When at is an **unhidden typed transition**:
  - $\forall < at, b' > \in From_b \in USS_B, < at, b' > \in From_b \in USS_A$, and
  - $\forall < at, val1, b' > \in ValFrom_b \in VSS_B, < at, val1, b' > \in ValFrom_b \in VSS_A$.
- vii) $Env_A =< C'_A, Rel_A >$ and $Env_B =< C'_B, Rel_B >$ are such that:
  - $visible(B) \setminus \{B\} \subseteq visible(A) \setminus \{A\}$
  - $\forall C_i, C_j \in C'_B$ such that $C_i \sqsubseteq C_j$ in Env_B, then $C_i \sqsubseteq C_j$ in Env_A.
  - $\forall C$ such that $B \sqsubseteq C$ in Env_B, then $A \sqsubseteq C$ in Env_A.
  - $\forall C$ such that $B \sqsupseteq C$ in Env_B, then $A \sqsupseteq C$ in Env_A.

3.3.3.4 Subclassing Examples

The list of examples that follow do not exhaustively identify interesting properties of the O-LSTSs semantics with regard to the **subclassing** relationship. However, the following examples do introduce some of the more important concepts. In particular, the examples illustrate the types of behaviour which are related by a **subclassing** relationship, and contrasts them with similar behaviours which are not related in this way.

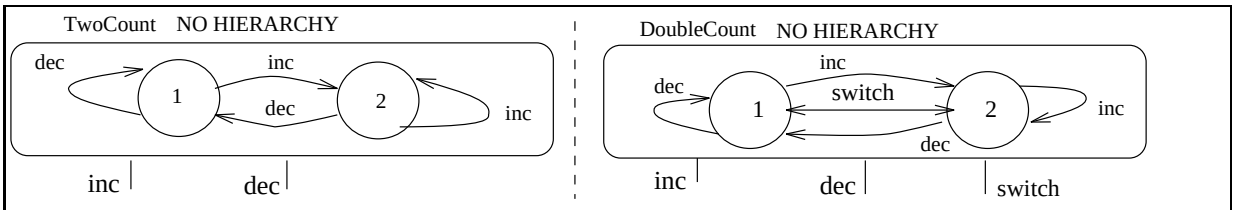


Figure 3.7: An Extension Example

Subclassing Example 1: Extension

Consider the O-LSTSs defined in figure 3.7.

Each of the O-LSTSs are '2-state machines'[11]. For simplicity, neither of the O-LSTSs has any **valued transitions**: they are said to be **non-responsive**. Furthermore, the O-LSTSs are defined in trivial environments ($< \{\}, \{\} >$). In this case, the **subclassing** relationship corresponds to some sort of structural (or topological) relationship between the internal representation of the O-LSTSs.

It is straightforward to prove, by checking the necessary and sufficient conditions of $\sqsubseteq$, that $DoubleCount \sqsubseteq TwoCount$ whilst $TwoCount \not\sqsubseteq DoubleCount$.

Proof: $DoubleCount \sqsubseteq TwoCount$, since conditions (i) to (vii) of the subclassing relationship are upheld:

- i) $\{1,2\} \subseteq \{1,2\}$
- ii) $\{inc, dec\} \subseteq \{inc, dec, switch\}$
- iii) $\{\} \subseteq \{\}$
- iv) $\{\} \subseteq \{\}$
- v) $\{\} \subseteq \{\}$
- vi) $\{< inc, 2 >, < dec, 1 >\} \subseteq \{< inc, 2 >, < dec, 1 >, < switch, 2 >\}$ and $\{< inc, 2 >, < dec, 1 >\} \subseteq \{< inc, 2 >, < dec, 1 >, < switch, 1 >\}$ and $VSS_{DoubleCount} = VSS_{TwoCount} = \{\}$
- vii) The environments of both O-LSTSs are trivially identical.

Proof: $TwoCount \not\sqsubseteq DoubleCount$, since
Condition (ii) of the subclassing relation is not fulfilled, since *switch* is an unhidden transition of $DoubleCount$ but *switch* is not an unhidden transition of $TwoCount$.

These O-LSTSs illustrate a particular relationship which we refer to as **extension**.

[11] An O-LSTS is said to be an 'n-state' machine iff the cardinality of its **state label set** equals n.

Definition: extension
A is an **extension** of B, written A **ext** $B \Leftrightarrow$
$A \sqsubseteq B$ and $(UAT(UTT_A) \cup VAT(VTT_A)) \subset (UAT(UTT_B) \cup VAT(VTT_B))$.

In other words, when A **ext** B, A offers all the attributes which B offers together with some additional attributes. When these additional attributes are ignored, every object in A behaves exactly like its corresponding object in B. The inverse of the **extension** relation is **restriction**:

Definition: restriction
A is a **restriction** of B, written A **res** $B \Leftrightarrow B$ **ext** A

Subclassing Example 2: Specialisation

The simple subclassing example in the O-LSTSD in figure 3.7 shows only a semantic relationship between unresponsive O-LSTSs with identical state sets. Both these restrictions are removed in the behaviours defined in figure 3.8.

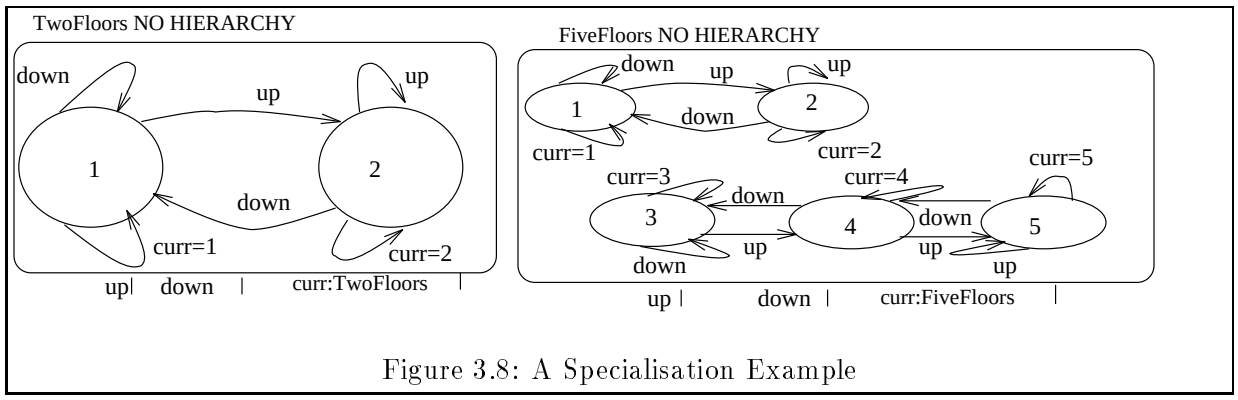


Figure 3.8: A Specialisation Example

Informally, *TwoFloors* specifies a lift system with 2 floors. The system can be requested to go up or down a floor. Also, it can respond with its current floor status when asked. It ignores requests to go up when it is on its top floor (in this case floor 2). Similarly, it ignores requests to go down on floor 1. *FiveFloors* specifies a lift system with 5 floors which cannot move between floors 2 and 3 (perhaps the lift system is broken). *FiveFloors* also ignores requests to go up and down whenever these movements are not possible. It should be clear that, by definition, $TwoFloors \sqsubseteq FiveFloors$. This is an example of **specialisation**.

Definition: specialisation
A is a **specialisation** of B, written A **spec** $B \Leftrightarrow A \sqsubseteq B$ and $States(A) \subset States(B)$.

Informally, if A **spec** B then B is partitioned into distinct sets of behaviour and A provides the behaviour of one or more, but not all, of these partitions. It is useful to be able to define a new class as a partition of an existing class. Such a class, which we refer to as a **partition class**, is specified by identifying a subset of the state set of the original class, provided this set is disjoint from the other state members. We say that an O-LSTS is **nonpartitionable** iff it has no **partition classes**. Like extension, specialisation has an inverse relation. It is called generalisation:

Definition: generalisation
A is a **generalisation** of B, written A **gen** $B \Leftrightarrow B$ **spec** A.

Extension and specialisation (and their inverse relations restriction and generalisation) are the only types of class relations which we consider in figure=Sem-Chp3/Figures. Different combinations of these relationships give rise to an unlimited number of possibilities.

3.3.3.5 Subclassing Guarantees Subtyping

We wish to show that $A \sqsubseteq B \Rightarrow A \leq B$ (The *TwoState* example in figure 4.8 has already shown that $A \leq B \not\Rightarrow A \sqsubseteq B$.) A simple example illustrates the relationship betwen subclassing and subtyping more clearly. Consider $Lift12s$ and $Lift15s$ defined in figure 3.9.

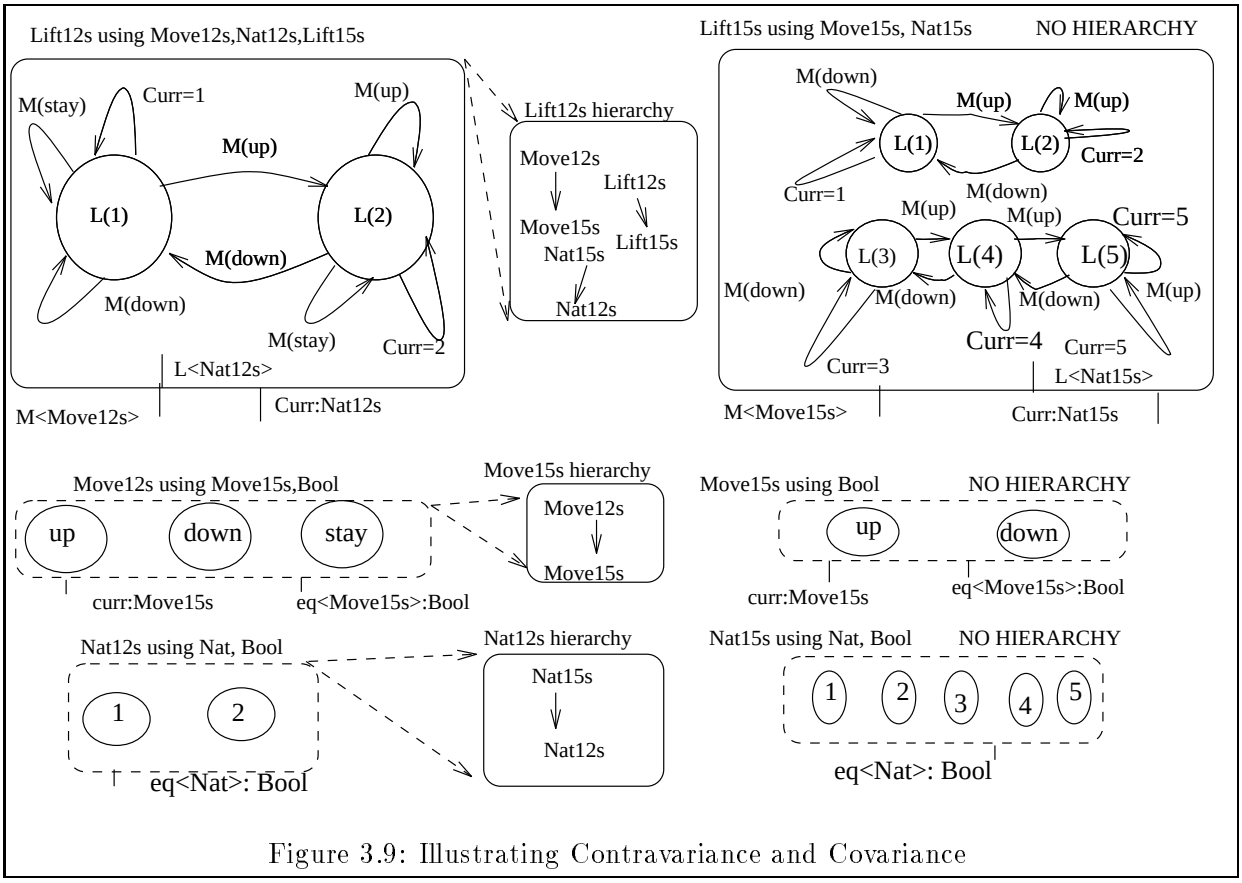


Figure 3.9: Illustrating Contravariance and Covariance

The O-LSTD representation has been extended in this example with two new constructs:

- The O-LSTSs with dotted boundaries are partial specifications in which the state-to-state transition sets are not shown. The partial specifications are used when we wish to represent only the interface of a class of behaviour: consequently, we refer to them as **interface** diagrams.
- The class hierarchies associated with each O-LSTS environment are represented by **class hierarchy diagrams** (the syntax and semantics of such diagrams is defined in section 3.3.3.6).

Consider the syntactic subtyping relationship between the different Lift O-LSTSs. It is clear, by the subtyping definition, that $Nat12s \leq\geq Nat15s$ and $Move12s \leq\geq Move15s$. Subsequently, we can

prove that $Lift12s \leq Lift15s$.

Proof: Lift12s is a subtype of Lift15s

The subtyping relationship is true since the rules of contravariance and covariance hold.

- contravariance:
  $M < Move15s > \in UTT_{Lift15s}$, $M < Move12s > \in UTT_{Lift12s}$ and $Move12s \geq Move15s$
- covariance:
  $curr : Nat15s \in VTT_{Lift15s}$, $curr : Nat12 \in VTT_{Lift12s}$ and $Nat12s \leq Nat15s$

Consider now the semantic subclassing relationship. It is obvious that if we assume that $Move15s \sqsubseteq Move12s$ and $Nat12s \sqsubseteq Nat15s$ then we can prove that $Lift12s \sqsubseteq Lift15s$, and as such the O-LSTSs are well-defined.

Proof: Lift12s is a subclass of Lift15s (given the above assuptions hold)

All the necessary and sufficient conditions for subclassing hold:

- The **state label set** of $Lift12s$ is a subset of the **state label set** of $Lift15s$, since $\{L(1), L(2)\} \sqsubseteq \{L(1), L(2), L(3), L(4), L(5)\}$
- The **unvalued unhidden actualised transition** set of $Lift15s$ is a subset of the **unvalued unhidden actualised transition** set of $Lift12s$, since $\{M(up), M(down), M(stay)\} \supseteq \{M(up), M(down)\}$and the **valued unhidden actualised transition** set of $Lift15s$ is a subset of the **valued unhidden actualised transition** set of $Lift12s$, since $\{curr\} = \{curr\}$
- The input parameters of $Lift12s$ are superclasses of the corresponding input parameters of $Lift15s$ in $Env_{Lift12s}$. This is true by the original assumption.
- The output parameters of $Lift12s$ are subclasses of the corresponding output parameters of $Lift15s$ in $Env_{Lift12s}$. This is true by our original assumption.
- The **state-to-state transitions** of $Lift15s$ are members of the **state-to-state transition** sets of $Lift12s$:
  $From_{L(1)_{Lift15s}} \subseteq From_{L(1)_{Lift12s}}$ and $ValFrom_{L(1)_{Lift15s}} \subseteq ValFrom_{L(1)_{Lift12s}}$
- The environment of $Lift12s$ is contained within the environment of $Lift15s$ since:
  - $visible(Lift15s) \backslash \{Lift15s\} = \{Bool, Nat, Nat15s, Move15s\} \subseteq visible(Lift12s) \backslash \{Lift12s\} = \{Move15s, Bool, Nat15s, Nat, Move12s, Nat12s, Lift15s\}$.
  - $Rel_{Lift15s} = \{\}$ and so there are no conditions to be met concerning the containment of subclassing relationships.

Proof: subclassing implies subtyping

Quite simply, removing conditions (i), (vi) and (vii) of the subclassing definition results in precisely the covariance and contravariance requirements for subtyping. Consequently, it is clear that subclassing is stronger than subtyping. More formally, $A \sqsubseteq B \Rightarrow A \leq B$.

3.3.3.6 Class Hierarchy Diagrams

The class hierarchy diagrams given in the previous example are an explicit statement of subclassing relationships which exist in the environment of an O-LSTS specification. The diagram is a graph of nodes and directed links between nodes. The nodes in the graph correspond isomorphically to the set of visible classes of the particular class, C say. We say that a path exists between nodes A and B iff there is a directed link from A to B, or $\exists$ a node C such that there is a directed link from A to C and there is a path from C to B. For every pair of visible classes, A and B say, related by the subclassing relationship $A \sqsubset B$ in Env_C, there is a path from the superclass node to the subclass node[12]. It is important to note that when one class is used by another, its class hierarchy is contained within the using class.

4.3.3.6 Fulfils — subclassing after syntactic relabelling

Strict conditions on the syntactic labelling of states and transitions are placed between classes related by the subclassing relationship and this can be a hindrance to re-use. Consider the simple examples of 2-state machines in figure 3.10.

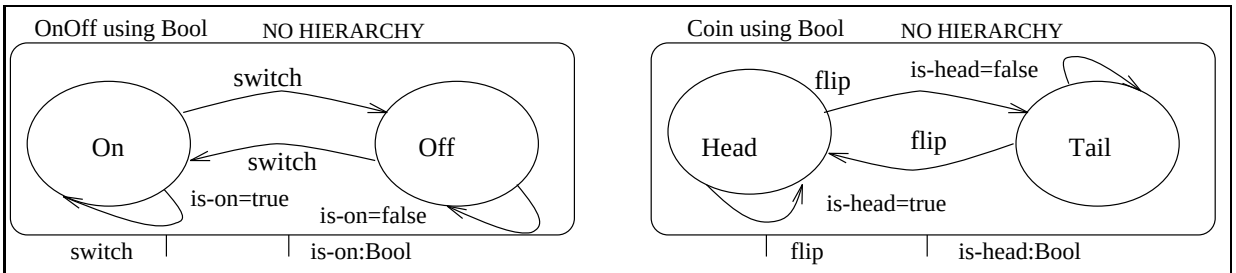


Figure 3.10: A Fulfils Example

These O-LSTSs are not related by the subclassing relationship. However, intuitively, they offer 'the same' behaviour. A simple syntactic relabelling of class names, state labels and transition labels 'transforms' either class into the other. When such a syntactic relabelling, using the transformation T on the class C say, results in a O-LSTS then the new class produced is labelled $T(C)$.

Definition: fulfils
A **fulfils** $B \Leftrightarrow \exists$ some syntactic relabelling transformation, T say, such that $A \sqsubseteq T(B)$.

This definition introduces one particular means of identifying one class of behaviour as a suitable candidate for providing the behaviour as specified by another class of behaviour. The examples in

[12]The symmetric subclassing relationships are not shown in the diagram.

figures 3.11 and 3.12 illustrate two particular types of fulfilment, namely transition reduction (**tr**) and state reduction (**sr**).

Fulfils Example 1: transition reduction

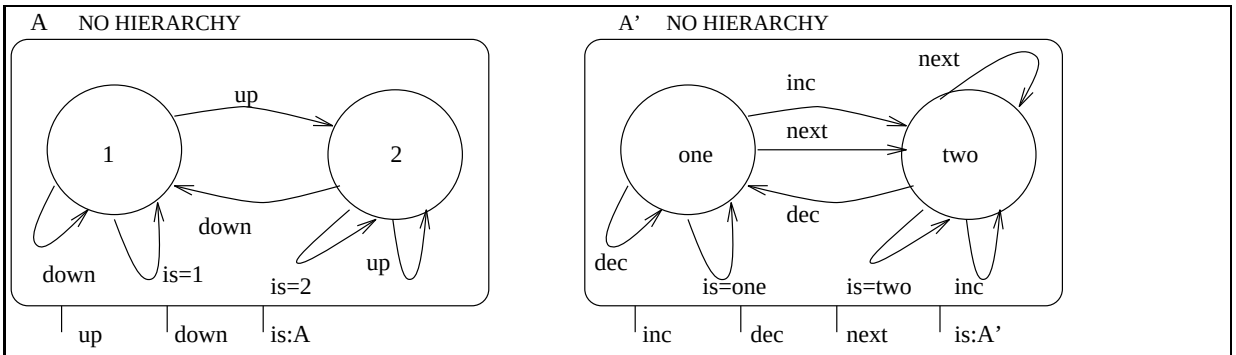


Figure 3.11: A Transition Reduction Example

Consider the O-LSTS in figure 3.11. A **fulfils** A' since $A \sqsubseteq T(A')$, where $T(A') = A$, $T(one) = 1$, $T(two) = 2$, $T(next) = up$, $T(inc) = up$, and $T(dec) = down$. The central idea is that the *next* and *inc* transitions in A' are equivalent (i.e. have the same effect). Consequently, the *up* transition in A can be used to fulfil both *next* and *inc* functionality.

This example illustrates how one class of behaviour simplifies the specification of the behaviour of another class. The O-LSTS A simplifies the specification of A' by reducing two equivalent transitions into one. This is formally defined as **transition reduction**.

Definition: transition reduction (tr)
A **tr** $B \Leftrightarrow A$ **fulfils** B and the cardinality of the union of the **typed transition sets** of A is less than the cardinality of the union of the **typed transition sets** of B.

Fulfils Example 2: state reduction

Consider the O-LSTSs defined in figure 3.12.

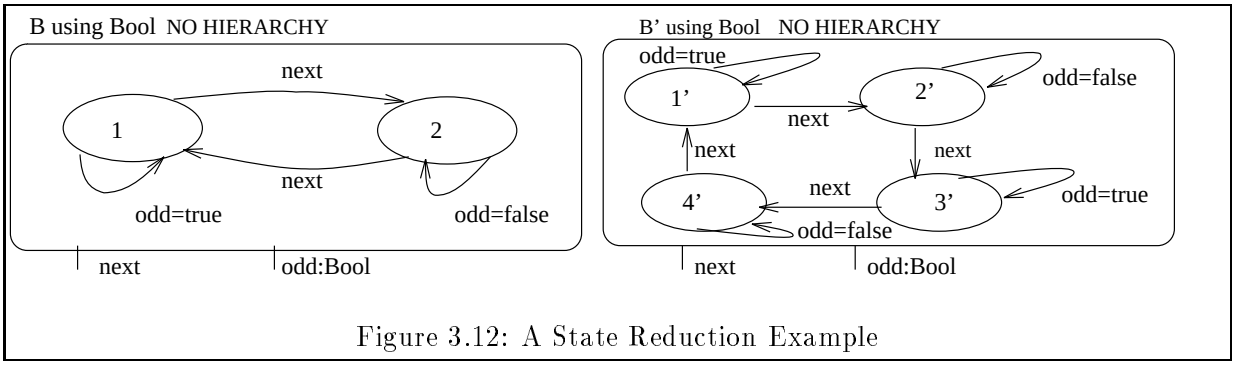


Figure 3.12: A State Reduction Example

B **fulfils** B' since $B \sqsubseteq T(B')$, where $T(B') = B$, $T(1') = 1$, $T(2') = 2$, $T(3') = 1$ and $T(4') = 2$. $T(B')$ is an O-LSTS which simplifies the specification of the behaviour of B' by reducing the number

of states in the system. In B' it is impossible to distinguish states $1'$ and $3'$ (and $2'$ and $4'$) through the external interface offered by the attribute set. The simplification which almalgamates equivalent states is called a state reduction.

Definition: state reduction (sr)
A **sr** $B \Leftrightarrow A$ **fulfils** B and the cardinality of the **state label** set of A is less than the cardinality of the **state label** set of B.

Fulfils: A Preview of Design Issues

The **fulfils** relationship is important when we consider re-use. When the behavioural requirements of a system (or system component) can be **fulfilled** by another already encoded component then it is sensible to re-use that implementation (after the appropriate syntactic relabelling). Note that the **fulfils** definition guarantees that a class always **fulfils** the behaviour of all of its superclasses. Consequently, subclassing provides a very particular kind of re-use facility. Confusion arises in object oriented terminology because subclassing is often thought of as a re-use mechanism rather than a relationship between classes which facilitates re-use.

Fulfils Example 3: Restructuring for design

The notion of fulfilment is important in design because it allows for the restructuring of class specifications whilst guaranteeing that the restructuring does not alter the requirements being defined. A simple example in figure 3.13 illustrates this.

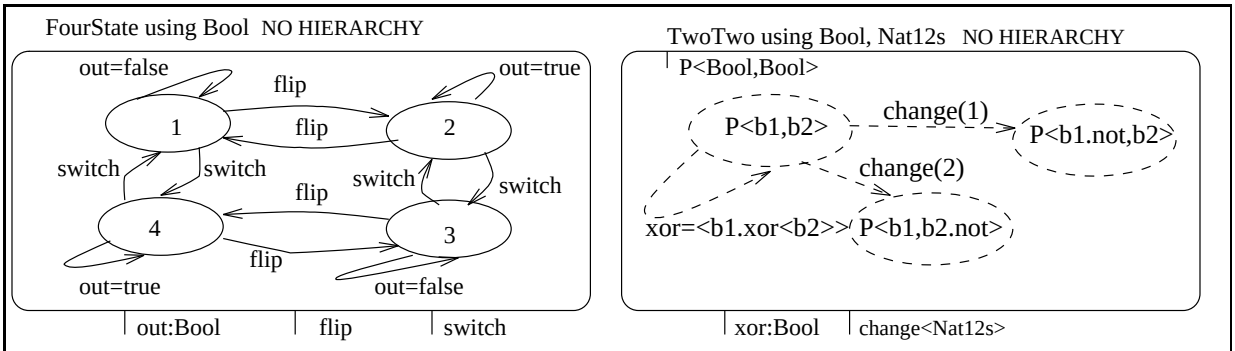


Figure 3.13: A Re-structuring Example

$TwoTwo$ **fulfils** $FourState$ since $TwoTwo \sqsubseteq T(FourState)$, where $T(out) = xor$, $T(flip) = change(2), T(switch) = change(1), T(1) = P(true, true), T(2) = P(true, false), T(3) = P(false, true)$, and $T(4) = P(false, false)$. $TwoTwo$ is a more structured specification than FourState. Adding meaningful structure to a specification (without changing the behaviour offered at the external interface) is an important aspect of design. Structure is fundamental to understanding — it encourages the re-use of pre-defined behaviours and the generation of re-usable behaviour. For example, the xor behaviour (provided by class $Bool$) is well understood and its re-use in $TwoTwo$ improves the specification.

In chapter 5, we consider design as a sequence of correctness preserving transformations. The *correctness* property between designs is related to the **fulfils** relationship in the analysis.

3.3.3.8 Inclusion

The **inclusion** example in figure 3.14 illustrates a form of re-use which is neither compositional nor subclassing. Nevertheless, this form of re-use is very common and effective. We refer to it as **inclusion**.

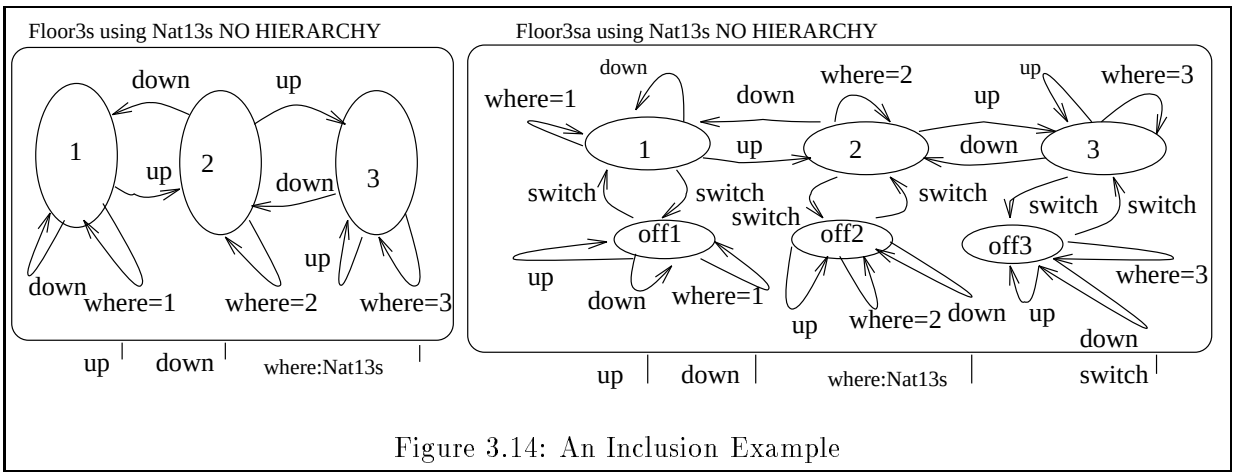


Figure 3.14: An Inclusion Example

$Floor3sa \not\sqsubseteq Floor3s$ since the 'off states' in $Floor3sa$ do not have any corresponding members in $Floor3s$. Also, $Floor3s \not\sqsubseteq Floor3sa$ since $Floor3s$ does not offer the *switch* attribute. Neither class can **fulfil** the other behaviour and neither class is **dependent** on the other. By ignoring the *switch* operator one can say that $Floor3s$ **specialises** $Floor3sa$. Clearly, it is advantageous to be able to define either one of these classes in terms of the other through some explicit re-use mechanism, which is based on the **inclusion** definition, below.

Definition: inclusion
B **includes** A , written B **inc** $A \Leftrightarrow$

- $\forall x \in US(B), From_x \in USS(B)$ is a subset of $From_x \in USS(A)$
- $\forall x \in US(B), ValFrom_x \in USS(B)$ is a subset of $ValFrom_x \in USS(A)$

This simple definition provides the foundation for a powerful purely syntactic re-use mechanism. To define A in terms of B it is necessary only to specify a subset of the **typed transition set** of B together with a subset of the **state label** set of B. To define B in terms of A it is necessary only to define new transitions and new states together with their associated behaviour.

3.3.4 O-LSTS Composition

An object in a class is represented by a **typed state label**. By definition, a **typed state label** is either unparameterised or parameterised. An unparameterised **typed state label** is represented by

a unique *state-constructor*. A parameterised **typed state label** is represented by a *state-constructor* followed by a list of parameter values.

An object which references an unparameterised state label is said to be **unstructured**. Such objects are not said to be composed from any component objects. An object which references a parameterised state label is said to be **structured**. A **structured** object is said to be composed from a set of component objects, which are precisely the objects corresponding to the parameter values. The optional boolean expression corresponds to an invariant property which collectively the components of the object must fulfil.

Reconsider the two O-LSTSs, $FourState$ and $TwoTwo$, specified in figure 3.13. The **state labels** of $FourState$ are unparameterised and consequently the objects which reference those labels are **unstructured**. Contrastingly, the **state labels** of $TwoTwo$ are parameterised — $\forall b1, b2 \in Bool, P(b1 : Bool, b2 : Bool) \in States(TwoTwo)$.

Composition is a relationship between objects. The notion can be extended to classes as follows. When all the **state labels** in an O-LSTS are represented by the same parameterised *state-constructor* then, since by definition the **parameter classes** are uniquely defined, the class is said to be **composed** from the **parameter classes** of the *state-constructor*. The *state-constructor* is said to define the **fixed structure** of the O-LSTS (class).

Definition: Composition

- **Object composition:** An object, O say, with corresponding **state label** *state-constructor*$(p_1, \ldots p_n)$ is said to be **composed** from $p_1, \ldots, p_n$, which are called the **components** of O.
- **State composition:** A class, C say, is said to be **composed** from classes $C_1, \ldots, C_n \Leftrightarrow \exists$ a state constructor, sc say, such that $\forall o \in States(C), \exists$ **state labels** $c_1, \ldots, c_n$ such that o is represented as $sc(c_1 : C_1, \ldots c_n : C_n)$.

When an object recieves a service request at one of its external attributes, it responds by returning a value and/or updating its internal state. A **structured** object achieves this functionality by requesting services of its components. These services may update the state of the components and/or return a result. With this in mind, it is now possible to formally define an object oriented interpretation of the internal processing that occurs when an object services a request. This interpretation is then used to formalise the representation of parameterised expressions in the O-LSTSD syntax. Consider the specification of two interacting stacks given in figure 3.15.

Two new pieces of syntax have been introduced:

- **Unspecified operations**

  The specifiers may not wish to define the result returned by a `pop` on an `empty Stack`. However, the implementors must provide the `pop` service for all `Stack` class members. The result of the `pop` operation must be a `Nat`. An unspecified member of the `Nat` class is represented by ∼`Nat`. This value is used to define the result returned by an `empty Stack` in response to a `pop` request. Unless otherwise specified, a service requested of an unspecified member always results in the

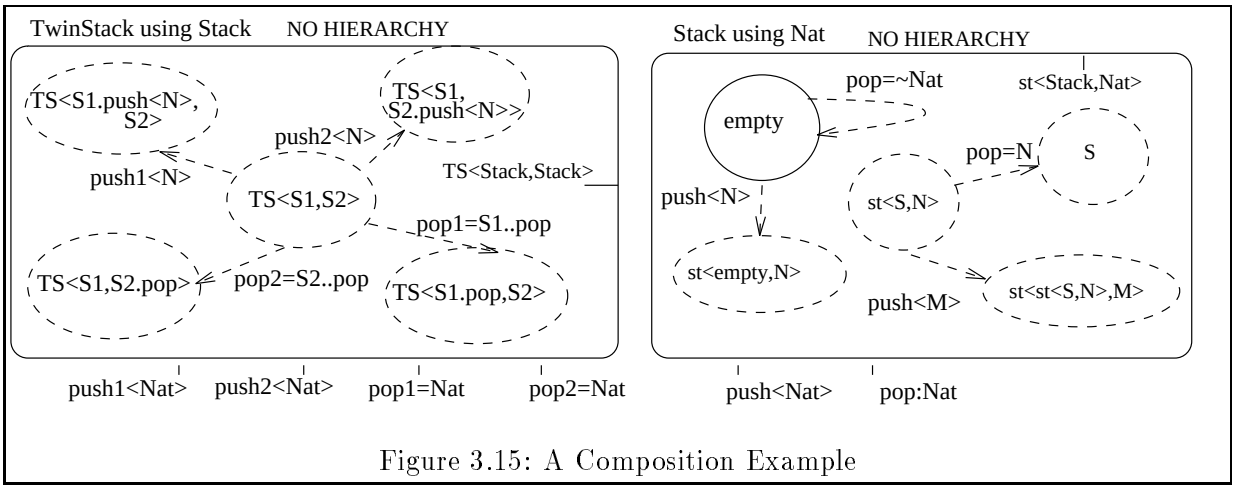


Figure 3.15: A Composition Example

unspecified member of the appropriate class. This default behaviour is implicit in every O-LSTS specification.

- **Using Component Services**

  A component object offers two 'responses' when it is sent a message request: it updates its internal state and/or replies with a result. The `pop` operation on a `Stack` results in the `Stack` replying with the last integer which had been pushed on, and updating its internal state by removing the top value. It is necessary to consider how the `pop` operation of a `Stack` component is used by the `TwinStack`. Service `pop1` on a `TwinStack` returns the value on the top of the first `Stack` component and updates the state of the component accordingly. Service `pop2` returns the value on the top of the second stack component and updates its state accordingly. Operations `push1` and `push2` are similarly defined.

  Two different pieces of syntax are applied:

  - The value returned in response to a valued service request, SR say, at an object O is represented by $O..SR$.
  - The new state of an object O after receiving a service request SR is represented by $O.SR$.

In section 4.2 we return to the notion of composition when using OO ACT ONE. More complex examples are considered, and the notion of an object 'restructuring' itself is examined. In particular, classes of objects with dynamic structure are investigated.

3.3.5 O-LSTS Configuration

Configuration is related to the notion of composition. Composition defines a hierarchical relationship between a client (containing object) and a server (component object). Configuration is a relationship between peer objects which are components of the same containing object. Components of the same containing object may, or may not, be configured.

Consider again the *TwinStack* O-LSTS (in figure 3.15). The class is composed from two components, both of which are `Stacks`. However, a `TwinStack` object never needs to use both components

to service one single request: `pop1` and `push1` use only the first `Stack` whilst `pop2` and `push2` use only the second component. In this case, the two `Stack` components are said to be **unconfigured**.

Consider the O-LSTS in figure 3.16.

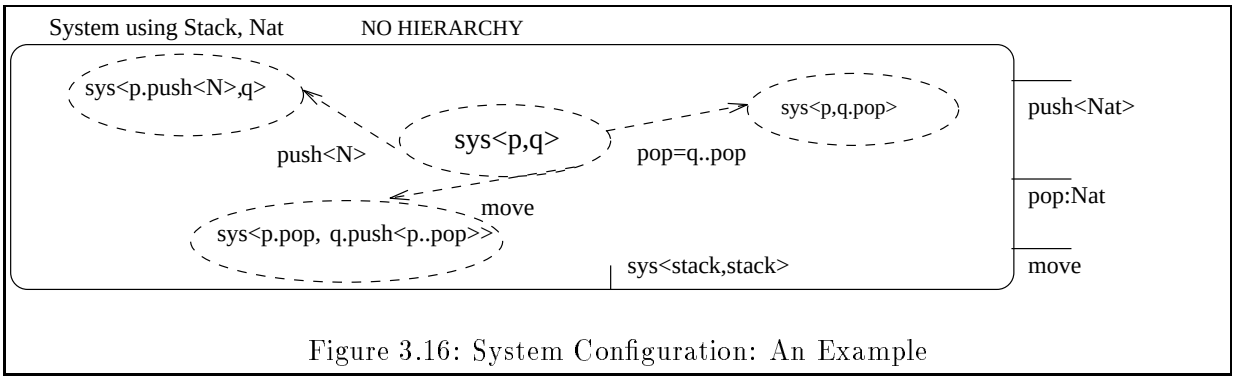


Figure 3.16: System Configuration: An Example

`System` is composed from two `Stack` components. The `push` transition puts the input parameter `Nat` value onto the first `Stack`. The `pop` transition removes the top `Nat` element from the second `Stack` and returns its value. The `move` transition transfers the top element of the first `Stack` onto the second `Stack`. The `move` transition is defined in terms of both components and so we say that the two `Stacks` are **configured** (by `move`). Note that configuration between objects does not necessarily imply interaction between the two components.

Configuration is formally defined below. The definition is based on the idea of one **state label expression** being used in the definition of another. The first expression is said to **depend on** the second. In particular, one object is said to **depend on** one of its components if the component is needed to fulfil an external service request.

Definition: Configuration

Objects A and B are **configured** (in object C) $\Leftrightarrow$

- A and B are **components** of C
- Either:
  - i) $\exists < uat, newstate > \in From_C$ such that
    $newstate$ **depends on** A and $newstate$ **depends on** B, or:
  - ii) $\exists < vat, val, newstate > \in ValFrom_C$ such that
    ($newstate$ **depends on** A or val **depends on** A) and ($newstate$ **depends on** B or val **depends on** B)

Definition: Dependence

A **state label expression**, SLE say, **depends on** another **state label expression**, sl say $\Leftrightarrow$ An expression of the form $sl.att$ or $sl..att$ appears in the representation of SLE.

3.3.6 Structure Diagrams

The internal structure of an object, i.e. an object's composition and configuration properties, is usefully represented in diagramatic form. For example, the O-LSTSs *System* and *TwinStack* are represented in the class structure diagrams in figure 3.17. The dotted circles represent classes of object. The class name is given above the circle. Attribute dependencies are shown as links joining the container class with the component classes[13] which it **depends on** to fulfil that particular attribute. Class structure diagrams are appropriate only when the class has a **fixed structure**.

Object structure diagrams differ from class structure diagrams in that actual objects are represented by solid circles and components are given concrete values. Figure 3.17 shows an object structure diagram for an element of the O-LSTS system which has two empty stacks as its internal components.

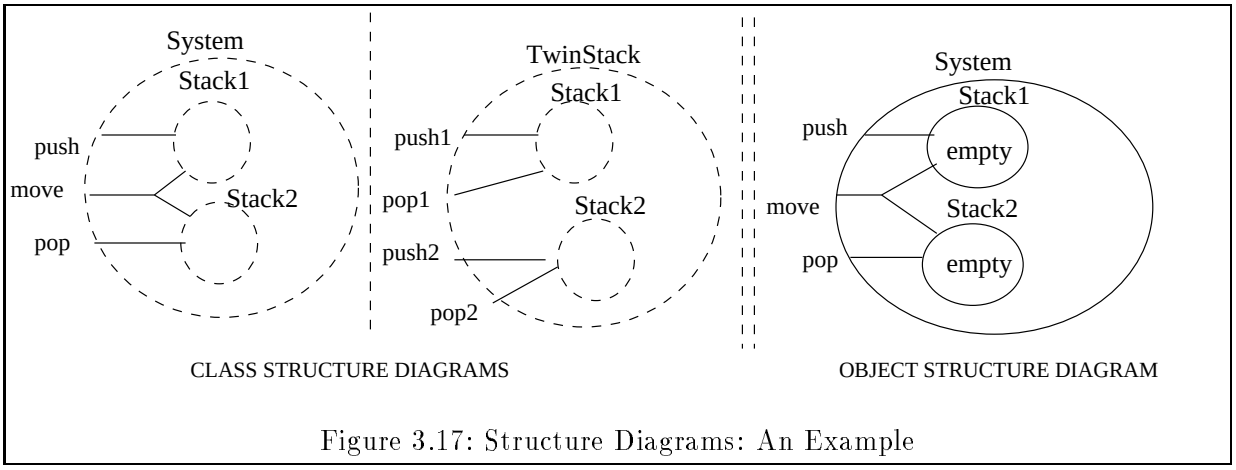


Figure 3.17: Structure Diagrams: An Example

3.4 OO ACT ONE: A Formal Object Oriented Analysis Language

3.4.1 Motivation

In this section we develop a concrete syntax for the specification of O-LSTS behaviour. The previous two sections illustrate how O-LSTSs can be defined using mathematical notation. This syntax is fine for the specification and illustration of simple behaviours. However, for the specification of more complex requirements, a *better* object oriented analysis language is required:

- An object oriented language should have an object oriented 'flavour'. It must be possible to reason about object oriented specifications in an object oriented conceptual framework. OO ACT ONE facilitates a more direct correspondence between the requirements being specified and the object oriented paradigm. It provides a number of high level mechanisms which syntactically sugar the O-LSTS model. The advantage of using OO ACT ONE over an informal object oriented language is that the underlying model is formally specified. Furthermore, OO ACT

[13] The components are identified by their class name and their index in the fixed structure. This index is used to distinguish components of the same class.

ONE specifications say *what* the system being specified should do rather than *how* it should do it.

- An O-LSTS specification must be statically analysed to check that all the necessary and sufficient conditions are met. In particular, the environment of an O-LSTS specifies a class hierarchy (set of subclassing relationships between O-LSTSs) which must be validated. Furthermore, there are 'typing constraints' on the visibility of state constructor (and state transition) parameters and result types which must be checked. A strict syntax for the specification of O-LSTS behaviour can help to make the 'type checking' easier to perform[14].
- The final goal of this chapter is to map our formal object oriented model onto ACT ONE. By defining a formal language which is similar in structure to ACT ONE, the translation to ACT ONE is simplified.

OO ACT ONE provides a practical means of specifying object oriented requirements in a formal framework.

3.4.2 The OO ACT ONE Syntax: Some Examples

Chapter 2 examines many of the practical issues in the design of a figure=Sem-Chp3/Figures language. It identifies the need for:

- A means of distinguishing between accessor, transformer and dual attributes.
- Comprehensive re-use facilities.
- A means of defining invariant properties which all class members must fulfil.
- A mechanism for hiding internal/local definitions or behaviour.
- A means of representing the structure of an object.
- A way of defining exceptions or unspecified behaviour.
- Explicit sub(super)classing mechanisms.

OO ACT ONE, a formal language which fufils all these needs, is best illustrated by the following list of examples. The first eight examples consider object based specifications, in which no subclassing relationships are explicitly defined. The final four examples consider in turn the four explicit (sub/super)class mechanisms, namely specialisation, generalisation, extension and restriction. We argue, in section 4.1, that these four mechanisms are sufficiently powerful for the general construction of class hierarchies during formal object oriented analysis. For simplicity, we are not yet concerned with the static analysis of the specifications which guarantees their correctness. All the example OO ACT ONE specifications that follow are well defined in the sense that they correspond to valid O-LSTSs.

[14]Perhaps 'type checking' is more accurately termed 'class checking'. However, since the notion of 'type checking' is pervasive in all areas of computing (even in object oriented development), we persist with this 'weaker terminology'.

Example 1: Classes Nat, Stack and System Revisited

The O-LSTS specifications of the O-LSTS `System` (see figure 3.16), the O-LSTSs `Stack` (see figure 3.15) and the unspecified `Nat` component provide good examples with which to illustrate the OO ACT ONE syntax. The `Nat` behaviour was not previously defined in O-LSTS form because its behaviour was not relevant to the example. For completeness, a simple `Nat` O-LSTS is defined in figure 3.18.

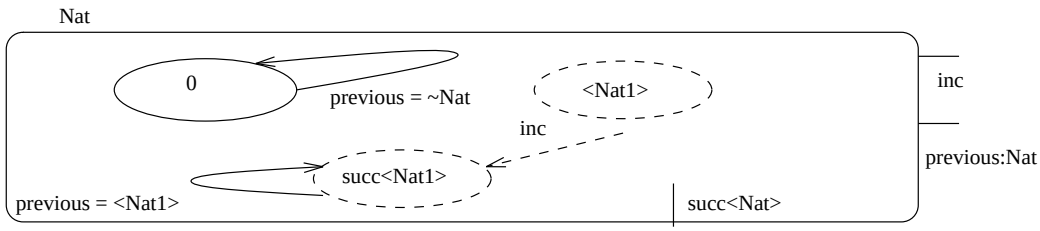


Figure 3.18: Specifying Natural Numbers: A Nat O-LSTS

The OO ACT ONE specification corresponding to the *Nat* O-LSTS is given below.

```
CLASS Nat OPNS
LITERALS: 0
STRUCTURES: succ<Nat> TRANSFORMERS: inc
ACCESSORS: previous -> Nat
EQNS
Nat1.inc = succ(Nat1); 0..previous = ~Nat; succ(Nat1)..previous = Nat1
ENDCLASS (*Nat*)
```

The following should be noted:

- Class `Nat` is not dependent on any other classes and consequently there are no class relationships defined between `Nat` and any other classes.
- The **state label expression** syntax and semantics is incorporated in OO ACT ONE.
- Variable parameters of a particular class (in equation definitions) are represented by the class name followed by an integer. In this way, unlike ACT ONE, the specifier does not need to explicitly 'type' the parameters of an expression. This elegantly concise syntax is unambiguous given the additional syntactic constraint that class names ending in an integer are not permitted (see 3.2.1.1).
- The unspecified member of a class is implicit in an OO ACT ONE specification.
- The OO ACT ONE syntax forces all operations and equations on a class to be defined in that class body. No new operation for a class can be defined in a different class.

The mapping between the `Nat` class and the *Nat* O-LSTS is very simple:

- $Env_{Nat} = < \{\}, \{\} >$, since `Nat` is defined independently of any other classes.
- The `LITERAL 0` corresponds to the **unparameterised typed state label** 0.

- The `STRUCTURE` operation `succ<Nat>` corresponds to the **parameterised typed state label** $succ(Nat)$.
- The `TRANSFORMER` operation `inc` corresponds to the **unvalued unparameterised typed transition** inc.
- The `ACCESSOR` operation `previous -> Nat` corresponds to the **valued unparameterised typed transition** $previous : Nat$.
- The equation `Nat1.inc = succ(Nat1)` corresponds to a parameterised set of **unvalued state-to-state transitions**: $< inc, succ(Nat1) > \in From_{Nat1}, \forall Nat1 \in US(Nat)$.
- The equation `0..previous = ~Nat` corresponds to the **valued state-to-state transition**: $< previous, \sim Nat, 0 > \in ValFrom_0$.
  (As `previous` is an accessor attribute, the state of the object servicing a `previous` request does not change.)
- The equation `succ(Nat1)..previous = Nat1` corresponds to a parameterised set of **valued state-to-state transitions**:
  $< previous, Nat1, Nat1 > \in ValFrom_{succ(Nat1)}, \forall Nat1 \in US(Nat)$.
- There are no **hidden** operations and consequently $HUTT$ and $HVTT$ in the corresponding O-LSTSs are empty sets.

The OO ACT ONE specification of the *Stack* O-LSTS uses the `Nat` behaviour. It is defined below.

```
CLASS Stack USING Nat OPNS
LITERALS: empty STRUCTURES: st<Stack, Nat>
TRANSFORMERS: push<Nat>
DUALS: pop -> Nat
EQNS
empty.push(Nat1) = st(empty,Nat1);
st(Stack1,Nat1).push(Nat2) = st(st(Stack1,Nat1),Nat2);
empty.pop = empty AND ~Nat;
st(Stack1,Nat1).pop = Stack1 AND Nat1
ENDCLASS (*Stack*)
```

This OO ACT ONE specification illustrates three new aspects of the OO ACT ONE syntax:

- `USING`: the `Stack` class is defined to 'use' the `Nat` class.
- `DUALS`: the `pop` operation is defined by an equation of the form: **state label expression1**.pop = **state label expression2** `AND` **state label expression3**. This defines a parameterised set of **valued state to state transitions**:
  $<$ *pop*, state label expression2, state label expression3$> \in ValFrom_{statelabelexpression1}$.
- `Paramaterised Attributes`: the operation `push<Nat>` corresponds to a **parameterised unvalued typed transition** $push(Nat)$. It has two associated equations:
  - `empty.push(Nat1) = st(empty, Nat1)`, which corresponds to a parameterised set of **unvalued state-to-state transitions**:
    $< push(Nat1), st(empty, Nat1) > \in From_{empty}, \forall Nat1 \in US(Nat)$.

- `st(Stack1, Nat1).push(Nat2) = st(st(Stack1, Nat1), Nat2)`, which corresponds to a
  parameterised set of **valued state-to-state transitions**:
  $< push(Nat2), st(st(Stack1, Nat1), Nat2) > \in From_{st(Stack1,Nat1)}$,
  $\forall Nat1, Nat2 \in US(Nat), Stack1 \in US(Stack)$.

The `Stack` specification is used by the `System` OO ACT ONE specification, defined below.

```
CLASS System USING Stack OPNS
STRUCTURES: sys<Stack, Stack>
TRANSFORMERS: push<Nat>, move
DUALS: pop -> Nat
EQNS
sys(Stack1,Stack2).push(Nat1) = sys(Stack1.push(Nat1), Stack2);
sys(Stack1,Stack2).move = sys(Stack1.pop, Stack2.push(Stack1..pop));
sys(Stack1,Stack2).pop = sys(Stack1, Stack2.pop) AND Stack2..pop
ENDCLASS (*System*)
```

Example 2: Grouping classes into modules

In ACT ONE, one type (a group of related sorts) can be defined using the group of sorts defined in another type using the `IS` construct. In other words, the `IS` construct in ACT ONE defines a relationship between types (not sorts). In OO ACT ONE we are more interested in the dependencies between classes of behaviour rather than the modules in which they are defined. However, it is still desirable to be able to re-use sets of related classes. As a compromise, we define a re-use facility between a class and a set of other classes which have been grouped together in a module definition. Module definitions are given before the set of class definitions which make up an OO ACT ONE specification. Module definitions are removed by a simple pre-processing of an OO ACT ONE specification. This syntactically substitutes the names of modules which are used by classes with the list of classes which are grouped within these modules. In other words, the modules in OO ACT ONE are simple syntactic sugaring — they do not extend the semantics of the O-LSTS model in any way. Consequently, they do not need to be considered in the mapping between OO ACT ONE and the O-LSTS model.

The classes grouped together in `module Degrees`, defined below, can be re-used by another class simply by listing the module name in the class header. For example, we can define a class 'Example' to use the classes Bool, Nat, Stack and all the classes in the module Degrees as follows: `Class Example USES Bool, Nat, MODULE Degrees, Stack`. The OO ACT ONE preprocessor removes the module definition and changes the `Example` class header to: `Class Example USES Bool, Nat, Joints, Singles, Stack`.

```
MODULE Degrees GROUPS Joints, Singles ENDMODULE (* Degrees *)
CLASS Joints USING Subject, Type OPNS
STRUCTURES: JointDegree<Subject, Type, Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
JointDegree(Subject1,Type1,Subject2)..studies(Subject3) =
(Subject1..eq(Subject3))..or(Subject2..eq(Subject3))
ENDCLASS (* Joints *)
CLASS Singles USING Subject OPNS
STRUCTURES: SingleDegree<Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
SingleDegree(Subject1)..studies(Subject2) = Subject1..eq(Subject2)
ENDCLASS (* Singles *)
```

Example 3: Invariant Properties

To illustrate the notion of an invariant property, consider a class similar to `Joints` except that it has no `Type` component. This class is defined as `Joints2` below.

```
CLASS Joints2 USING Subject OPNS
STRUCTURES: JointDegree<Subject, Subject>
ACCESSORS: studies<Subject> -> Bool
EQNS
JointDegree(Subject1, Subject2)..studies(Subject3) =
(Subject1..eq(Subject3)).or(Subject2..eq(Subject3))
ENDCLASS (* Joints2 *)
```

`JointDegree(Maths, Maths)` is a member of the `Joints2` class provided that `Maths` is a literal member of the `Subject` class. This may not be desirable behaviour since we may require, in the system that we are modelling, that a joint degree consist of two different subjects. To specify this condition we have two options:

- Explicitly list all joint degree combinations which are valid.
- Define an invaraint property on the structure `JointDegree` to specify that the first component cannot be the same as the second component.

The second option is better since an explicit statement of the invariant property improves the understandibility of the specification. Furthermore, using an invariant property follows the principle of encapsulation and makes the specification simpler to extend. For example, if the `Subjects` class is to be extended to include a new literal then this change should be possible without affecting the classes which use the `Subjects` class. This is not possible with the first option, in which the principle of encapsulation has to be broken for the behaviour of the degree class to be well defined. The `JointDegree` structure is respecified in class `Joints3` to incorporate the new invariant property.

All the members of a class now correspond to the literals and the structure expressions whose component values fulfil the relevant structure invariant(s) (if there are any). Invariant properties

```
CLASS Joints3 USING Subject OPNS
STRUCTURES: JointDegree<Subject, Subject>
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: JointDegree(Subject1, Subject2) REQUIRES Subject1..neq(Subject2)
EQNS
JointDegree(Subject1,Subject2)..studies(Subject3) =
(Subject1..eq(Subject3)).or(Subject2..eq(Subject3))
ENDCLASS (* Joints3 *)
```

introduce the possibility of 'run time' errors in an execution model. For example, consider an extension to the `JointDegrees` class in which a new transformer operation allows either of the `Subject` fields to be changed. Now, a transformer service may result in a new state which does not fulfil the invariant property. In this case the behaviour of the resulting object is undefined.

The invariant above is termed a **structure invariant** because the invariant property is defined in terms of properties of the components of a structure of the class. It is also often desirable to be able to define an invariant on a whole class rather than a structure in a class. The syntax of such a **class invariant** is illustrated by the `MathsJoints` specification below. Class invariant properties are required to be true for all literal values (this is checked by a static analysis). This requirement makes the specification of class invariants much easier to transform (during pre-processing) into a set of structure invariants. It also makes the concept of class invariant much easier to understand — why define a literal value which does not fulfil an invariant property?

```
CLASS MathsJoints USING Subject OPNS
STRUCTURES: JointDegree(Subject, Subject)
ACCESSORS: studies<Subject> -> Bool
INVARIANTS: MathsJoints1..studies(Maths)
EQNS
JointDegree(Subject1, Subject2)..studies(Subject3) =
(Subject1..eq(Subject3))..or(Subject2..eq(Subject3))
ENDCLASS (* MathsJoints *)
```

Note that the class invariant mechanism is simply syntactic sugaring for defining sets of structure invariants. For example, the invariant `MathsJoints1..studies(Maths)` can be re-written as: `JointDegree(Subject1,Subject2) REQUIRES (Subject1..eq(Maths))..or(Subject2..eq(Maths))`. As the external attributes of a structured class are defined in terms of the external attributes of its components, a class invariant is just a more concise way of expressing a set of **structure invariants**. Consequently, in mapping OO ACT ONE to the O-LSTS model we must consider only how to map structure invariants: **class invariants** are removed by a simple pre-processing of an OO ACT ONE specification.

Structure invariants correspond to boolean conditions of **conditioned parameterised typed state labels** in the O-LSTS model, represented as **state label expressions** of **type** Bool. Invariant properties, in OO ACT ONE, depend on the visibility of a `Bool` class (with members `true` and `false`).

Example 4: An includes (re-use) mechanism

The advantages of having a purely syntactic re-use mechanism is illustrated by the `Floor3s` and `Floor3sa` OO ACT ONE specifications, below.

```
CLASS Floor3s USING Nat13s OPNS
LITERALS: 1,2,3
TRANSFORMERS: up, down
ACCESSORS: where -> Nat13s
EQNS 1.up =2; 2.up=3; 3.up=3; 1.down=1; 2.down=2; 3.down=3;
1..where =1; 2..where=2; 3..where=3
ENDCLASS (* ----------------- Floor3s ----------------- *)
CLASS Floor3sa USING Nat13s OPNS
LITERALS: 1,2,3,off1,off2,off3
TRANSFORMERS: up, down, switch
ACCESSORS: where -> Nat13s
EQNS 1.up =2; 2.up=3; 3.up=3; off1.up=off1; off2.up=off2; off3.up=off3; 1.down=1; 2.down=2;
3.down=3; off1.down=off1; off2.down=off2; off3.down=off3; 1.switch = off1; 2.switch = off2;
3.switch = off3; off1.switch = 1; off2.switch = 2; off3.switch = 3; 1..where =1; 2..where=2;
3..where=3; off1..where =1; off2..where =2; off3..where =3
ENDCLASS(* Floor3sa *)
```

The O-LSTSs corresponding to these OO ACT ONE specifications are defined in the **inclusion** example in figure 3.14. Although there is no subclassing relationship between these two behaviours, it is clear that it is beneficial to be able to define either class in terms of the other. More generally, a mechanism for including some of the operations and equations from one class in a new class definition is required. The `INCLUDES` mechanism is illustrated by the `Floor3s'` specification below.

```
CLASS Floor3s' OPNS INCLUDE FROM Floor3sa
LITERALS: 1,2,3
TRANSFORMERS: up, down
ACCESSORS: where
ENDCLASS (* Floor3s *)
```

This new definition re-uses part of the specification of `Floor3sa`. The operations that are re-used have to be explicitly listed. The equations for these operations do not need to be listed. The equations for the included transformer, accessor and dual operations on the included literals and/or structures are implicit in the new specification. In this example, it is therefore not necessary to define any additional equations. Note that `Floor3s'` is not defined to use `Floor3sa`. The includes mechanism does not copy the classes which are used by the class being included.

The includes mechanism identifies an operation of a class and 'copies' its operation and equation definitions into the new class. (It is not a direct copy since all the occurrences of the old class name have to be replaced by occurrences of the new class name in the included definition part of the new class.) The `includes` mechanism is further sugared to give an `includesall` mechanism which states that all the operations and equations of the specified class are copied into the new specification. The inclusion mechanisms are a sort of MACRO expansion facility. They do not extend the semantics of

the ADT language. OO ACT ONE specifications are preprocessed to remove all include directives.

Example 5: Internal Operations

The specification of internal (or hidden) operations is common to many object oriented programming languages (see chapter 6). It is useful to be able to define attributes of a class which are available only to the class in which they are defined (i.e. attributes which are not part of the external interface). These attributes are then used to help specify the external behaviour. In OO ACT ONE, accessor, dual and transformer operations can be declared as `HIDDEN`. This is illustrated by the specification of a simple store of natural numbers, below.

```
CLASS Store USING Nat, Bool OPNS
LITERALS: empty STRUCTURES: st<Store, Nat>
TRANSFORMERS: add<Nat>
ACCESSORS: average -> Bool, sum -> Nat (* HIDDEN *), size -> Nat (* HIDDEN *)
EQNS
empty..average = ~Nat; empty..sum = 0; empty..size = 0;
st(Store1,Nat1)..average = (st(Store1,Nat1)..sum)..div(st(Store1,Nat1)..size);
st(Store1, Nat1)..sum = Nat1.+(Store1..sum); st(Store1, Nat1)..size = 1.+(Store1..sum)
ENDCLASS (* Store *)
```

In this specification, the `HIDDEN` operations are defined to simplify the definition of the external operation `average`. The `HIDDEN` operations have a direct correspondence with the elements of $HUTT$ and $HVTT$ (the **hidden typed transitions**) in the O-LSTS model. The definition of **state label expressions** constrains hidden operations to being used only inside the class in which they are defined. This constraint must be checked during the static analysis of OO ACT ONE specifications.

Example 6: Preconditions

A parameterised equation definition, defined on a class or a structure of a class, can be preconditioned by a **state label expression** of **type Bool**. This expression must be parameterised on a (non strict) subset of the parameters in the equation definition (Such an expression can be represented in general form as `Pre(p1,..., pn)`, where `p1` to `pn` are the parameters of the parameterised equation.). Preconditions are a powerful mechanism for simplifying specifications and improving their understandibility. The syntax of the precondition mechanism in OO ACT ONE is illustrated by the specification of class `Queue`, below.

The `Queue` example shows the syntax for defining structure preconditions on dual operations. The syntax for defining the results of preconditioned transformer and accessor operations is the same as the first and second parts, respectively, of the dual syntax (the parts separated by `AND`). The mapping between preconditioned equations and the O-LSTS model is straightforward. It is detailed in appendix A.

```
CLASS Queue USING Nat, Bool OPNS
LITERALS: empty STRUCTURES: Q<Queue, Nat>
TRANSFORMERS: add<Nat>
ACCESSORS: is-empty -> Bool (*HIDDEN*)
DUALS: rem -> Nat
EQNS
empty..is-empty = true; Q(Queue1, Nat1)..is-empty = false;
empty.add(Nat1) = Q(empty, Nat1);
Q(Queue1, Nat1).add(Nat2) = Q(Q(Queue1, Nat1),Nat2);
empty.rem = empty AND ~Nat;
Queue1..is-empty =>
Q(Queue1, Nat1).rem = empty AND Nat1 OTHERWISE Q(Queue1.rem, Nat1) AND Queue1..rem
ENDCLASS (*Queue*)
```

Example 7: Generic (Parameterised) Classes

Genericity is a powerful mechanism. A generic class does not specify a behaviour in the sense that it corresponds to one O-LSTS specification: it acts as a template (or structure) from which other classes of behaviour can be constructed. A generic class is parameterised on the classes which it uses. An instance of a generic class is created through an actualisation of the class parameters. The OO ACT ONE generic mechanism (like many aspects of the syntax) is based on the corresponding mechanism in ACT ONE (with a few syntactic differences to reinforce the object oriented flavour of the language). Classes of behaviour which are defined as instances of generic classes can be transformed into classes which are not defined generically.

It should be noted that genericity is not a subclassing mechanism. Users of other object oriented languages, in which subclassing is not formally defined, often argue that a generic class is a superclass of its instances. However, we argue that a generic class is not a class in its own right, it is a template for creating classes. It is possible to define a generic class such that there are subclassing relationships between instances, depending on the actual parameterisation of the instantiated classes. In figure=Sem-Chp3/Figures we believe that genericity and subclassing are two very different concepts and as such they should be kept distinct. Consequently, we choose to define OO ACT ONE so that generic classes cannot be related by 'sets of subclassing' relationships. Thus, we do not allow generic classes to be specialised, generalised, extended or restricted (see examples 9 to 13), although instances of these classes can be used in this way. The genericity syntax is illustrated by the `Pair` class below.

It is not necessary to further expand on the semantics of generic classes: they are handled in the same way as generic types in ACT ONE. In fact, they map directly onto the ACT ONE generic construct when we generate an ACT ONE model from the OO ACT ONE requirements.

Example 8: Renaming

It is often useful to be able to define a new class to exhibit the same semantic behaviour as another but to have a different syntactic representation. In OO ACT ONE we allow one class to be defined by renaming the operation string identifiers of another class. For example, consider the specification

```
GENCLASS Pair USING Bool, GCLASS elementA, GCLASS elementB
GCLASS elementA GOPNS
GACCESSORS: eq<elementA> -> Bool
ENDGCLASS (* elementA *)
GCLASS elementB GOPNS
GACCESSORS: eq<elementB> -> Bool
ENDGCLASS (* elementB *)
OPNS
STRUCTURES: P<elementA, elementB>
ACCESSORS: eq1<elementA> -> Bool, eq2<elementB> -> Bool
TRANSFORMERS: set1<elementA>, set2<elementB>
EQNS
P(elementA1, elementB1)..eq1(elementA2) = elementA1..eq(elementA2);
P(elementA1, elementB1)..eq2(elementB2) = elementB1..eq(elementB2)
ENDCLASS (* Pair *)
```

of the class `TwoNats` given below.

```
CLASS TwoNats RENAMES NatPair
STRUCTURES: P WITH A2Nat
TRANSFORMERS: set1 WITH change1, set2 WITH change2
ENDCLASS (*TwoNats*)
```

The structure operation `P` and the transformer operations `set1` and `set2` are renamed in the new `TwoNats` class specification. By default, all operations which are not renamed retain their original names. The renaming is removed by a simple pre-processing.

Examples 1 to 8 show the object based mechanics of OO ACT ONE (i.e. the object oriented mechanics without class relationships). OO ACT ONE restricted to this syntax is called OB ACT ONE[15]. All valid OB ACT ONE specifications are valid OO ACT ONE specifications. The OB ACT ONE syntax allows for the formal specification of:

- Classes of behaviour which are protected behind strict interfaces.
- The composition of predefined classes into new (more complex) classes of behaviour.
- Invariant properties which all class members must uphold.
- Internal (hidden) operations.

It is necessary to extend the object based mechanisms with subclassing facilities. In particular, we wish to be able to define a new class to be a subclass (or superclass) of an already existing class and be guaranteed that the corresponding O-LSTSs are related by the formal subclassing relationship. Four such mechanisms are provided in OO ACT ONE, namely extension, restriction, specialisation and generalisation. These correspond to the relationships between O-LSTSs which are defined in section 3.3.3. These four mechanisms, together with a mechanism which combines specialisation and extension, provide the only means of explicitly defining class relationships in OO ACT ONE.

[15]Object Based ACT ONE.

Example 9: Extension

In the O-LSTS semantics, a class A is defined to be an extension of a class B when A offers the behaviour of B together with some additional behaviour. Furthermore, all the members of A must be members of B. This notion of **extension** has a corresponding mechanism in OO ACT ONE. Reconsider the O-LSTS specifications of $TwoCount$ and $DoubleCount$ given earlier in figure 3.7. The OO ACT ONE specification of class `TwoCount` is given below.

```
CLASS TwoCount OPNS
LITERALS: 1,2 TRANSFORMERS: inc, dec
EQNS 1.inc =2; 2.inc =2; 1.dec =1; 2.dec =1
ENDCLASS (*TwoCount*)
```

The `EXTENDS` mechanism can be used to explicilty define the class `DoubleCount` as a subclass of `TwoCount`. This new class, `DoubleCount2` say, is defined in OO ACT ONE below. The O-LSTSD which corresponds to `DoubleCount2` also given in figure 3.19; this should be compared with `DoubleCount` in figure 3.7. Note that the environment of the new class records the explicit subclassing relationship between `DoubleCount2` and `TwoCount`.

```
CLASS DoubleCount USING TwoCount
EXTENDS TwoCount WITH OPNS
TRANSFORMERS: switch
EQNS
1.switch = 2; 2.switch = 1
ENDCLASS (* DoubleCount *)
```

Class `TwoCount` corresponds to the O-LSTS specification given in figure 3.7.

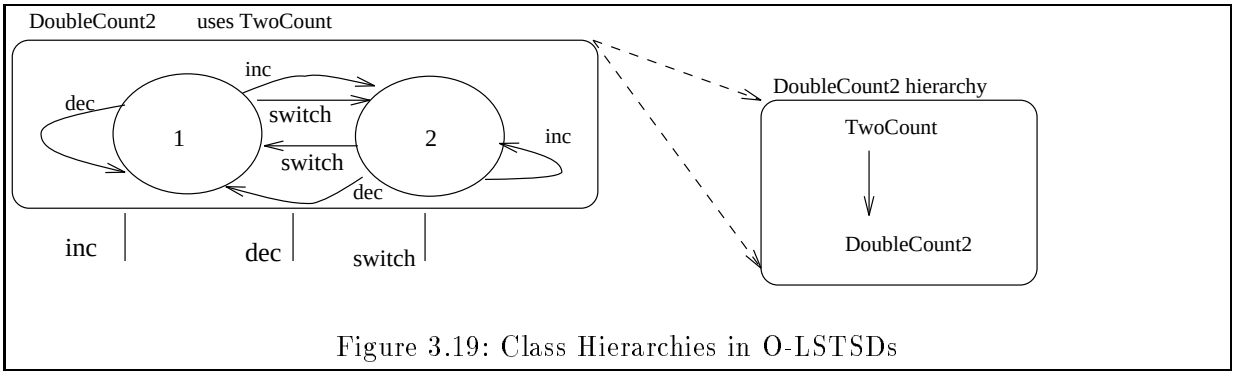


Figure 3.19: Class Hierarchies in O-LSTSDs

The environment of $TwoCount$ is trivial: $Env_{TwoCount} =< \{\}, \{\} >$. `DoubleCount2` corresponds to the O-LSTS specification in the figure 3.19 with a non-trivial environment:
$Env_{DoubleCount2} =< \{TwoCount\}, \{< DoubleCount2, TwoCount >\} >$. Note that the same object based behaviour could have been defined using the `INCLUDESALL` mechanism. In this instance, the O-LSTS 7-tuple corresponding to the OO ACT ONE specification would be the same, but the environment of the system would no longer explicitly acknowledge the subclassing relationship.

The syntactic constraints of the `EXTENSION` mechanism prevent new literals or structures from being defined on a class extension. Furthermore, invariant properties cannot be strengthened or weakened (i.e. the sets of invariants which are defined for the original class are precisely the set of invariants defined for the new extended subclass).

The explicit definition of the subclassing relationship `DoubleCount2` $\sqsubseteq$ `TwoCount` is significant in the semantics of **state label expressions**. Any **state label expression** in which an attribute parameter of class `TwoCount` is expected can accept a parameter of the class `DoubleCount2`.

Example 10: Restriction

Restriction is the inverse relation of extension. Given the original `DoubleCount` specification we would like to be able to define a class `TwoCount` as a superclass (restriction) of `DoubleCount`. A new class can be defined as a restriction of another class by defining a strict subset of the accessor, transformer and dual operations of the class being restricted. Given the following OO ACT ONE specification of class `DoubleCount`, it is possible to define a new class `TwoCount2` as a restriction of `DoubleCount`. This is illustrated below.

```
CLASS DoubleCount
DoubleCount OPNS
LITERALS: 1,2
TRANSFORMERS: inc, dec, switch
EQNS
1.inc =2; 2.inc =2; 1.dec = 1; 2.dec =1; 1.switch = 2; 2.switch = 1;
ENDCLASS (* DoubleCount *)
CLASS TwoCount USING DoubleCount RESTRICTS DoubleCount TO OPNS
TRANSFORMERS: inc, dec
ENDCLASS (* TwoCount *)
```

Example 11: Specialisation

Reconsider the O-LSTSs in figure 3.9. `Nat12s` is defined explicitly (by its environment) to be a subclass of `Nat15s`. These behaviours were only partially specified in the previous section so, for completeness, the actual behaviours are defined below.

The `SPECIALISES` construct requires that the new class must explicitly identify the literals and structures which it has in common with its superclass. (The static analysis of an OO ACT ONE specification must verify that these members form a valid partition of the original class — see 3.4.3).

Example 12: Generalises

Generalisation is the inverse of specialisation. In figure 3.9, `Move12s` **gen** `Move15s`. The two classes of behaviour are defined below.

The `Move12s` class specification illustrates how one class which generalises another offers the complete behaviour of the other class as a partition of itself. We extend the generalises mechanism to

```
CLASS Nat15s USING Nat, Bool OPNS
LITERALS: 1,2,3,4,5
ACCESSORS: eq<Nat> -> Bool
EQNS
1..eq(Nat1) = succ(0)..eq(Nat1); 2..eq(Nat1) = succ(succ(0))..eq(Nat1);
3..eq(Nat1) = succ(succ(succ(0)))..eq(Nat1);
4..eq(Nat1) = succ(succ(succ(succ(0))))..eq(Nat1);
5..eq(Nat1) = succ(succ(succ(succ(succ(0)))))..eq(Nat1)
ENDCLASS (* Nat15s *)
CLASS Nat12s USING Nat15s SPECIALISES Nat15s TO OPNS LITERALS: 1,2
ENDCLASS (* Nat12s *)
```

```
CLASS Move15s USING Bool OPNS
LITERALS: up,down
ACCESSORS: eq<Move15s> -> Bool, curr -> Move15s
EQNS up..eq(up) = true; up..eq(down) = false;
down..eq(up) = false; up..eq(up) = true;
up..curr = up; down..curr = down
ENDCLASS (* ----------------- Move15s ----------------- *)
CLASS Move12s USING Move15s GENERALISES Move15s WITH OPNS
LITERALS: stay
EQNS stay..eq(up) = false; stay..eq(down) = false; stay..curr = ~Move15s
ENDCLASS (* Move12s *)
```

allow one class to be defined as a generalisation of a group of classes. For example, the `stay` member of `Move12s` can be defined as a distinct (though very limited) class of behaviour in its own right. This new class, `JustStay` say, can be generalised with `Move15s` to give the `Move12s` behaviour. This group generalisation is illustrated below.

```
CLASS JustStay USING Move15s OPNS
LITERALS: stay
ACCESSORS: eq<Move15s> -> Bool, curr -> Move15s
EQNS stay..eq(up) = false; stay..eq(down) = false; stay..curr = ~Move15s
ENDCLASS (* ---------------------- JustStay ---------------------- *)
CLASS Move12s USING Move15s, JustStay GENERALISES Move15s, JustStay
ENDCLASS (* Move12s *)
```

Subclassing: contravariance and covariance mechanisms

The classification examples given above do not consider subclassing relationships in which the rules of contravariance and covariance are exploited: all the parameter types are defined by default to be the same in the subclasses as they are in the superclasses. It is necessary to override this default in three different cases:

- i) The class parameters of a structure operation of a subclass may be defined as subclasses of the corresponding parameters in the superclass.

- ii) The result of a valued transition in a subclass may be defined as a subclass of the corresponding result type in the superclass.
- iii) The parameters of a transition in a subclass may be defined as superclasses of the corresponding parameters in the superclass.

These non-default options are illustrated by the O-LSTS specifications of `Lift12s` and `Lift15s` in figure 3.9. All three non-default options are taken by `Lift12s` as a subclass of `Lift15s`. These two classes of behaviour are not connected by an extension or specialisation relationship alone. The behaviour of `Lift12s` is a combination of an extension and specialisation of `Lift15s`. In OO ACT ONE we make this relationship explicit by defining `Lift12s SPECIALISES AND EXTENDS Lift15s`. The complete OO ACT ONE specifications corresponding to the two O-LSTSs are given below.

```
CLASS Lift15s USING Move15s, Nat15s OPNS
STRUCTURES: L<Nat15s>
TRANSFORMERS: M<Move15s>
ACCESSORS: curr -> Nat15s
EQNS L(1).M(up) = L(2); L(2).M(up) = L(2);
L(3).M(up) = L(4); L(4).M(up) = L(5); L(5).M(up) = L(5);
L(1).M(down) = L(1); L(2).M(down) = L(1);
L(3).M(down) = L(3); L(4).M(down) = L(3); L(5).M(down) = L(4);
L(1).curr =1; L(2).curr =2; L(3).curr =3
ENDCLASS (* ------------------ Lift15s ------------------ *)
CLASS Lift12s USING Move12s, Nat12s SPECIALISES AND EXTENDS Lift12s TO OPNS
STRUCTURES: L<Nat12s>
TRANSFORMERS: M<Move12s>
ACCESSORS: Curr<Nat12s>
EQNS L(1).M(stay) = L(1); L(2).M(stay) = L(2); L(3).M(stay) = L(3)
ENDCLASS (* Lift12s *)
```

This example illustrates the only combination of class relationships which can be defined by one mechanism: `SPECIALISES AND EXTENDS` . A `SPECIALISES AND EXTENDS` B states that it is possible to define a class C such that A `EXTENDS` C and C `SPECIALISES` B. Using the combination mechanism means that the class C does not need to be explicitly defined. OO ACT ONE does not define any other 'combination mechanisms': in our experience `SPECIALISES AND EXTENDS` is the only combination mechanism which is used as often as the other singular mechanisms (when provided). Furthermore, it is the only combination mechanism which is straightforward to statically analyse.

A Note On Invariant Properties

It is important to note that all invariant properties are inherited by a subclass from its superclasses.

3.4.3 Static Analysis of OO ACT ONE: Syntax and Semantics

OO ACT ONE specifications are statically analysed as shown in figure 3.20.

Appendix B1 examines the preprocessing of an OO ACT ONE specification for the removal of syntactic sugar. Appendix B2 explains the static analysis of OO ACT ONE specifications which

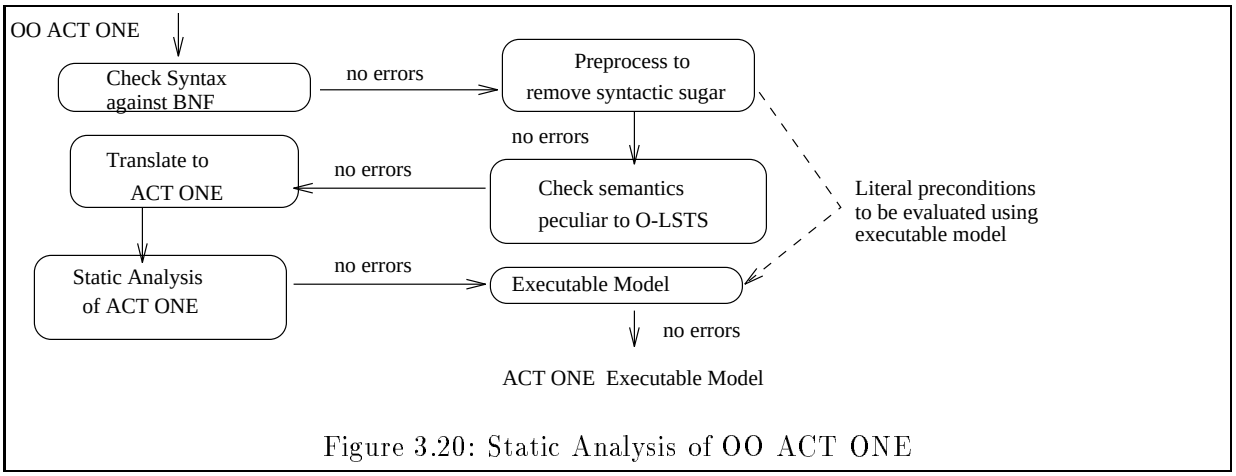


Figure 3.20: Static Analysis of OO ACT ONE

cannot be done by translating the specification to ACT ONE and letting the ACT ONE static analysis tools check the new code.

3.5 An ACT ONE Execution Model for O-LSTS Specifications

3.5.1 The Advantages of Using ACT ONE

Translating an O-LSTS specification, written in OO ACT ONE, into ACT ONE has three advantages:

- It more formally relates the object oriented notions captured in the O-LSTS semantics with the ADT concepts of type, sort, operation, equation, expression and value.
- ACT ONE has a number of associated tools for the static analysis of syntax and semantics and the evaluation of expressions. These tools can be used to complete the static analysis of the OO ACT ONE specification from which the ACT ONE was developed, and to test the dynamic behaviour being modelled.
- ACT ONE forms a part of full LOTOS, which combines the ADT specifications within a process algebra as a single coherent framework. This thesis proposes to use LOTOS as an object oriented design language. The step from formal object oriented analysis to formal object oriented design is made simpler by choosing ACT ONE as the foundation upon which our figure=Sem-Chp3/Figures language is modelled.

3.5.2 Reviewing the ACT ONE Terminology

In ACT ONE, a type specification may define a number of different sorts. Every sort corresponds to a set of terms, with each term representing a particular value of that sort. Equations define equivalences between terms which represent the same value. Each equivalence class of terms represents one value and members of the same equivalence class are identified by that value. The member of the equivalence class which is used to represent the value of the class is normally taken to be the term which all the other terms are 'evaluated to' when the re-writes (as defined by the equations) are applied.

An operation relates 0,1 or more input sorts to a result sort. Operations are term generators. Operations with no input terms are called **literals**. These are commonly used to represent the value of the equivalence class of terms to which they belong. Furthermore, they are the base terms from which other terms can be generated.

3.5.3 An Overview of the OO ACT ONE → ACT ONE Translation

The translation from OO ACT ONE to ACT ONE is best described in stages. Preprocessing has removed OO ACT ONE syntactic sugar and so there are just five object based elements of the syntax to consider:

- `CLASSES`, which are the fundamental behaviour building blocks.
- The `USING` mechanism, which defines a dependency between classes.
- `OPNS`, which define the external interface and set of members for each class.
- `EQNS`, which define the dynamic behaviour of each class member.
- `INVARIANTS`, which define properties fulfilled by members of the classes in which they are defined.

In addition, there are five explicit classification mechanisms to be mapped to ACT ONE. Appendix C examines the semantics of the mapping from OO ACT ONE to ACT ONE.

Additonal Object Based Mechanics

The ACT ONE mechanisms for defining the behaviour of an object which changes state and returns a value in response to a service request are called the **dual mechanics**. Other object based 'mechanics' are required to define the behaviour of the implicit unspecified members of OO ACT ONE classes. Further, additional operations (internal tests) are defined to simplify the specification of the object oriented execution model. The mechanics defined in each of these cases is as follows:

- **Dual Mechanics**
  Values of an ACT ONE sort corresponding to members of an OO ACT ONE class have two types of representation, namely singular and dual. Singular representations are literals or structure expressions. Dual representations of a class `C` are pairs of values whose first element corresponds to a singular representation of the class `C` and whose second element corresponds to any sort value which represents a member of the **result type** of an accessor or dual operation of `C`. For every **result type**, `D` say, of a class `C` there is an operation `dualCD : C,D -> C` which is used to construct the corresponding dual representation. The external attributes (as defined by the accessor, dual and transformer operations) of dual representations of values in `C` are defined as the attribute operations applied to the first element of the dual expresssion.

  Singular representations are generated by transformer operations on singular or dual representations. Dual representations are generated by accessor or dual operations. In OO ACT ONE we represent the newstate of an object, `obj` say, after servicing a request `req` say, by the **state label expression** `obj.req`. Similarly, we represent the value returned by an accessor or dual

operation as `obj..req`. **State label expressions** in OO ACT ONE are mapped into ACT ONE as follows:

- `obj.req` → `.(req(obj))`
- `obj.req(p1,...,pn)` → `.(req(obj,p1,...,pn))`
- `obj..req` → `ClassResult(req(obj))`, where `Class` is the **result type** of `req`.
- `obj..req(p1,...,pn)` → `ClassResult(req(obj,p1,...,pn))`, where `Class` is the sort generated from the **result type** of `req`.

This ACT ONE representation of OO ACT ONE **state label expressions** requires the specification of two additional operations:

- `.:C -> C` is defined for every class `C`.
- `DResult: C -> D` is defined for every **result type** `D` in class `C`.

- **Unspecified Values**
  Implicit in every OO ACT ONE specification are the unspecified literal values of each class. For example, ∼C represents the unspecified literal value of class C. In translating to ACT ONE, we generate an operation `unspecC: -> C` for every sort `C`. All operations on this value are defined to return the unspecified value of the appropriate class (by default). Static analysis of an OO ACT ONE specification determines when such defaults are overridden by the specifier.
- **Internal Tests**
  For every class `C` we define an operation `CRep: C -> Bool` which returns true if the input parameter of the `CRep` operation is represented in singular form. This internal test operation is used to simplify the specification of the object oriented 'mechanics'.

3.5.3.2 Example Object Based Behaviours in ACT ONE

The ACT ONE specifications that follow result from translating object based behaviour as specified in OO ACT ONE. (The code that is listed is slightly different from the ACT ONE code that is produced because the object oriented features which allow this class to be defined as a superclass of a new class are not included. We consider such object oriented concerns in section 3.5.3.3.) The three examples are used to illustrate different aspects of the object based properties specified by the ACT ONE code.

Example 1: `Nat` behaviour

Consider the OO ACT ONE `Nat` class specified in example 1 of section 3.4.2 (and its corresponding O-LSTSD in figure 3.18). The ACT ONE code is given below.

There are a number of things to note about this specification:

- Although class `Bool` is not specified as being used by class `Nat` in the OO ACT ONE requirements, a Boolean type with sort Bool is used in the ACT ONE specification of sort `Nat`. Every sort generated from an OO ACT ONE specification is defined in terms of boolean behaviour (it is necessary for defining the internal object oriented mechanisms). Consequently, the type

```
TYPE Nat IS Boolean SORTS Nat OPNS
0: -> Nat (* Literal*)
succ:  Nat -> Nat (* Structure *)
inc:  Nat -> Nat (* Transformer *)
previous:  Nat -> Nat (* Dual Accessor Nat *)
unspecNat:  -> Nat
.:  Nat -> Nat
NatResult:  Nat -> Nat
dualNatNat:  Nat, Nat -> Nat
NatRep:  Nat -> Bool
EQNS FORALL Nat1, Nat2:  Nat
OFSORT Nat
inc(Nat1) = succ(Nat1); inc(unspecNat) = unspecNat;
inc(dualNatNat(Nat1,Nat2)) = inc(Nat1);
previous(0) = dualNatNat(0,unspecNat); previous(unspecNat) = unspecNat;
previous(succ(Nat1)) = dualNatNat(succ(Nat1), Nat1);
previous(dualNatNat(Nat1, Nat2)) = previous(Nat1);
NatRep(Nat1) => .(Nat1) = Nat1; .(dualNatNat(Nat1, Nat2)) = Nat1;
NatResult(dualNatNat(Nat1,Nat2)) = Nat2;
OFSORT Bool
NatRep(0) =true; NatRep(succ(Nat1)) = true; NatRep(unspecNat) = true;
NatRep(dualNatNat(Nat1,Nat2)) = false
ENDTYPE (* Nat *)
```

`Boolean` with sort `Bool` is an integral part of the resulting ACT ONE code. This class is specified to exhibit the well understood behaviour of booleans (all the normal operations are available as transformer operations) together with the unspecified value `unspecBool`.

- The ACT ONE specification has an intuitively object oriented style. Ignoring the additional object oriented mechanics (which are generated in the same way for all behaviours), we have a clear and concise correspondence between the ACT ONE and the OO ACT ONE from which it was generated.
- The list of variables after the `forall` clause in the ACT ONE code is defined to exactly match the variable parameters used in the equation definitions.
- The translation to ACT ONE produces very inefficient code. However, efficiency is not important at this theoretical stage of development.

Example 2: `System` behaviour

Consider the OO ACT ONE `System` class also specified in example 1 of section 3.4.2 (and its corresponding O-LSTSD in figure 3.16). In this example we assume the ACT ONE code for class `Stack` has been generated in sort `Stack` defined in the type of the same name. The ACT ONE code which is generated from the class `System` specification is given below.

This specification shows quite clearly the way in which the components of the `System` (i.e. the `Stacks`) are used through their external interfaces alone to provide the external behaviour of their containing object.

```
TYPE System IS Boolean, Stack SORTS System OPNS
sys:  Stack, Stack -> System (* Structure *)
push:  System, Nat -> System (* Transformer *)
move:  System -> System (* Transformer *)
pop:  System -> System (* Dual Nat *)
unspecSystem:  -> System
.:  System -> System
NatResult:  System -> Nat
dualSystemNat:  System, Nat -> System
SystemRep:  System -> Bool
EQNS FORALL System1:  System, Nat1, Nat2:  Nat, Stack1,Stack2:  Stack
OFSORT System
push(sys(Stack1, Stack2), Nat1) = sys(.(push(Stack1, Nat)), Stack2);
push(unspecSystem, Nat1) = unspecSystem;
push(dualSystemSystem(System1,Nat1), Nat2) = push(System1, Nat2);
move(sys(Stack1, Stack2)) =
sys(.(pop(Stack1)), .(push(Stack2, NatResult(pop(Stack1)))));
move(unspecSystem) = unspecSystem;
move(dualSystemSystem(System1,Nat1), Nat2) = move(System1, Nat2);
pop(sys(Stack1, Stack2)) =
dualSystemNat( sys(Stack1, .(pop(Stack2))), NatResult(pop(Stack2)));
pop(unspecSystem) = unspecSystem;
pop(dualSystemSystem(System1,Nat1)) = pop(System1);
SystemRep(System1) => .(System1) = System1;
.(dualSystemNat(System1, Nat1)) = System1;
NatResult(dualSystemNat(System1,Nat1)) = Nat1;
OFSORT Bool
SystemRep(sys(Stack1, Stack2)) = true; SystemRep(unspecSystem) = true;
SystemRep(dualSystemNat(System1,Nat1)) = false
ENDTYPE (* Nat *)
```

Example 3: Preconditions in the specification of `Queue` behaviour

The ACT ONE specification of the sort `Queue` given in Appendix C2 is used to illustrate the mapping of preconditioned expressions to ACT ONE. A less important feature of this example is the mapping of a hidden operation.

3.5.3.3 Translating Object Oriented Requirements

An Overview

To translate object oriented requirements specified in OO ACT ONE to ACT ONE, it is necessary to group together classes of behaviour which are related by subclassing relationships into one type definition in ACT ONE. Consider the OO ACT ONE specifications of `Lift12s` and `Lift15s` defined at the end of section 3.3.3.5 (and their corresponding O-LSTSDs given in figure 3.9). These classes of behaviour are translated into the framework of ACT ONE code given below in the type definition of `Lift12sRoot`.

```
TYPE Lift12sRoot IS Move12sRoot, Nat12sRoot
SORTS Lift12s (* using Move12s, Nat12s, Lift15s *)
Lift15s (* using Move15s, Nat15s:  superclass Lift12s *)
OPNS ...EQNS ...ENDTYPE
TYPE Move12sRoot IS Boolean
SORTS Move12s (* using Bool, Move15s *)
Move15s (* using Bool:  superclass Move12s *)
OPNS ...EQNS ...ENDTYPE
TYPE Nat15sRoot IS Boolean, Nat
SORTS Move12s (* using Bool, Nat *)
Nat12s (* using Bool, Nat:  superclass Nat15s *)
OPNS ...EQNS ...ENDTYPE
```

The three classes of object oriented behaviour that are modelled in this ACT ONE header are `Lift12s, Move12s` and `Nat15s`, i.e the root classes of the separate trees in the class hierarchy of `Lift12s`. It is possible to execute a dynamic model of the other non-root classes but there is no guarantee that the environments of the these classes are correctly specified. To guarantee that the environment of a class, `C` say, is correctly modelled in ACT ONE, it is sufficient to restrict the classes listed in the OO ACT ONE code to be only those classes visible to `C`.

Class Hierarchy Mechanisms

There are two important aspects of modelling object oriented behaviour:

- **Polymorphism**:

  In the object oriented paradigm, subclassing means that a member of one class is also a member of each of its superclasses. Consequently, anywhere a member of one of its superclasses is used by an object, all the members of the subclass must also be able to be used. This is inclusion polymorphism. In OO ACT ONE there is one general instance of this rule, namely in a parameterised operation we require that an actual parameter is a member of the parameter class or a member of a subset of the parameter class. The paramaterised operations which need to be considered are: structures, accessors, transformers and duals.

  When generating ACT ONE code from an OO ACT ONE specification we model polymorphism by defining all the parameterised operations on any combination of valid parameter. This is done by defining coercion routines between any two classes related by a subclassing relationship. Subclass parameters are coerced into being the corresponding members of the required superclass. The coercion operations are defined as follows: for every pair of classes, `C1` and `C2` say, such that $C1 \sqsubseteq C2$ in the environment of the class being modelled, then there is an operation `C1toC2: C1 -> C2` defined in the type of the class to which these classes are rooted. The operation `C1toC2` is defined by a set of equations which equate all literal values of `C1` to the same literal values of `C2`. Further, all subclass `structure` expressions are coerced to superclass `structure` expressions by applying a suitable coercion to the component values.

- **Inheritance:**

  The existence of a subclassing hierarchy suggests that there is a duplication of behaviour between

superclasses and subclasses. This duplication can be taken advantage of when creating an object oriented model of OO ACT ONE requirements in ACT ONE. The behaviour defined by an operation applied to a member of a class is always, whenever possible, inherited from a superclass[16] of that class. In ACT ONE we encode this inheritance by specifying explicit root definitions for operations which are not inherited from superclasses. These root operations (defined as `ClassOpn` where `Class` is the name of the class in which the behaviour is rooted and `Opn` is the name of the operation) are then used in the subclass definitions to avoid duplication. Coercion plays an important part in this inheritance mechanism since it is now also necessary to be able to coerce superclass values into subclass values.

The additional machinery required to model this object oriented behaviour further complicates the ACT ONE code. In general, the ACT ONE code is between four and twenty times larger[17] than the OO ACT ONE from which it was generated. An example of the ACT ONE code arising from the translation of object oriented properties in OO ACT ONE is given in Appendix C3.

3.5.4 Static Analysis of ACT ONE

The static analysis of ACT ONE code generated from OO ACT ONE guarantees certain correctness properties of the underlying O-LSTS model. The ACT ONE static analyser checks the types of all parameters in equation definitions (their visibilty and compatibility). It also checks the correctness of all **state label expressions** in the O-LSTS model specified using OO ACT ONE. Thus, by translating to ACT ONE, the most difficult static analysis is performed by an already existing tool set. One problem is that static analysis errors identified in the ACT ONE code have to be translated back into meaningful OO ACT ONE errors. It should be clear that, although such a mechanism is not formulated in this work, providing an object oriented interpretation of these ACT ONE errors is not a difficult task.

3.5.5 Evaluating Act One Expressions: An Execution Model for OO ACT ONE

To model the behaviour of an object in response to a message request at its interface it is necessary only to evaluate an ACT ONE expression. For example, reconsider the `System` behaviour considered earlier in section 3.5.3.2. To model the effect of a `pop` request at the object represented by `sys(Stack1, Stack2)` we evaluate the ACT ONE expression `pop(sys(Stack1,Stack2))`. The result of this expression evaluation is a dual representation. The first element of this pair of values is the newstate of the object, the second element represents the value returned by the `pop` operation.

To model the dynamic behaviour of an object over a period of time it is necessary to create a feedforward loop of expression evaluations in which the result of an expression evaluation is used as the **server** of the next expression to be evaluated. Such loops of behaviour are represented in **event**

[16] When two superclasses offer the same operation we are guaranteed that the behaviour defined in both superclasses is the same. Consequently, when a choice is available, an arbitrary decision is made as to which superclass a subclass inherits from.

[17] Size in this case is an approximation for the number of operations and equations defined.

diagrams (see below). By taking this simple view of object oriented behaviour, the static ADT model is given a dynamic object oriented interpretation.

3.5.6 Event Diagrams

An event diagram for the behaviour of a `System` object is given in figure 3.21, together with the corresponding **event trace**.

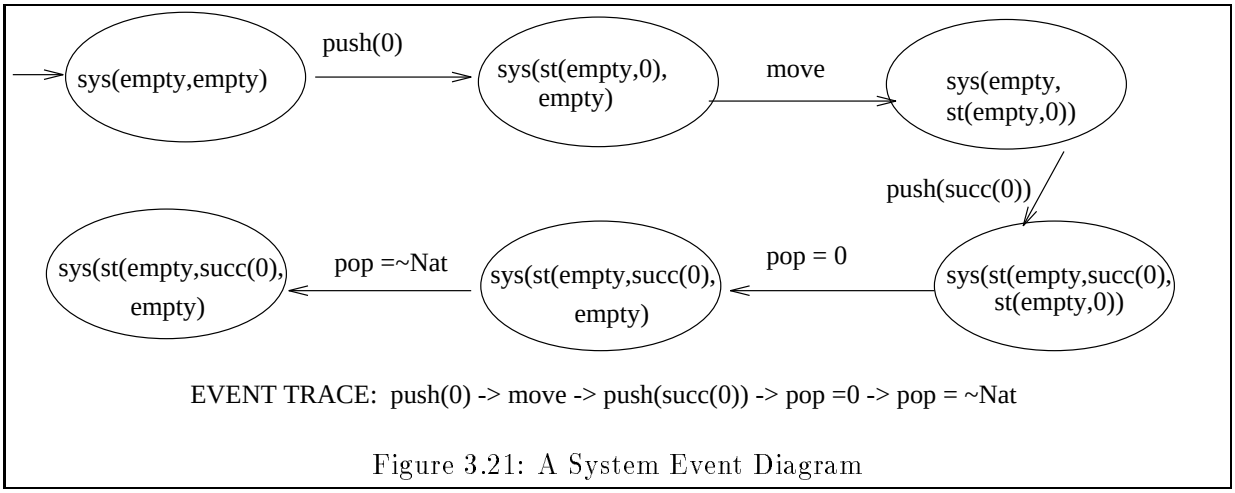


Figure 3.21: A System Event Diagram

An O-LSTS event diagram is simply an an abstraction of the whole O-LSTS(D) in which only a particular set of connected states are represented. The initial state of the system is represented by the only state in the diagram with an incoming arrow which is not rooted in another state. The final state of the system is represented by the only state in the diagram without an outgoing arrow. The sequence of transitions which the system goes through is called an **event trace**. Such traces are common in process algebras: they specify possible behaviours of a process (or system). An O-LSTS gives rise to a peculiar set of **event traces** because of the constraint that an O-LSTS system must always be able to fulfil all of its service requests at all times during its life. The property which distinguishes different instantiations of the same O-LSTS specification are the sequences of values which they return during execution.

Chapter 4

Formal Object Oriented Analysis: The Practical Issues

This chapter examines the more practical issues which arise during object oriented analysis and requirements capture (when using OO ACT ONE). The discussions which are undertaken assume that the underlying formality of OO ACT ONE is well understood. The object oriented-ness of OO ACT ONE, and its suitability to the task of requirements capture and analysis, is no longer being considered. Rather, we proceed to investigate more general object oriented analysis issues, using OO ACT ONE to illustrate the points being made.

The structure of this chapter is as follows:

- **Section 4.1: Subclassing**
  This section examines the role of subclassing during object oriented analysis. Two different types of subclassing hierarchy are identified, namely those which offer multiple inheritance features and those which do not. Polymorphism and dynamic binding are then considered. The need to differentiate between explicit and implicit subclassing relationships is emphasised. Then, the concept of abstract superclass is given a more rigorous formulation. Finally, the classic polymorphism problem of heterogenous structures is explored.
- **Section 4.2: Composition**
  This section examines the notion of composition and its fundamental role in object oriented analysis. Emphasis is given to distinguishing composition from configuration and interaction. Two different types of compositional structures are introduced: dynamic and static. Then a pure style of OO ACT ONE specification is defined to model the object oriented notion of persistency. Finally, this section investigates the modelling of shared objects and timing properties during analysis.
- **Section 4.3: Other Object Oriented Analysis Issues**
  This section examines a potpourri of other object oriented issues: concurrency, nondeterminism, communication models, exception handling, the active/passive categorisation of objects, persistency and class routines concerned with creation and configuration.

- **Section 4.4: Reviewing The OO ACT ONE Specification Language**
  Section 4.4 reviews the OO ACT ONE specification language and asks if it is a *good* analysis technique when judged by the criteria put forward in chapter 2.
- **Section 4.5: The Practicalities of Building a Formal Model**
  This section begins by listing a set of critreria which an analysis method (as opposed to set of models) must fulfil. Then it defines a *skeleton* method for building a formal object oriented requirements model using OO ACT ONE. This *skeleton* method is shown to place emphasis on: re-use of pre-defined behaviours, recording of problem domain structure and improving problem domain understanding. After the method is formulated, some more general questions concerning analysis decisions, which influence the style of a specification produced using this method, are put forward. Then guidelines are given for making changes to the requirements models. Finally, a list of general analysis principles are proposed, which analysts can rely on to help make difficult development decisions.
- **Section 4.6: figure=FormAnal-Chp4/Figures and Object Oriented Design**
  This section introduces the process of going from analysis to design. It argues that the structure of a requirements specification is fundamental to the initial design of a solution to the problem. It is argued that executable requirements models are advantageous to object oriented development. Finally, as a preview of chapter 5, the notion of correctness preserving transformation is introduced.

4.1 Subclassing

4.1.1 Categorising Class Hierarchies

When one class is defined as a subclass of another it is said to **inherit** features of its superclass. Object oriented languages support either single inheritance or multiple inheritance. We argue that multiple inheritance is necessary for modelling customer understanding.

4.1.1.1 Single Inheritance

In single inheritance models, classes are allowed at most one parent (direct superclass). The class hierarchy diagram in figure 4.1 illustrates the type of hierarchy which arises when such a restriction is enforced.

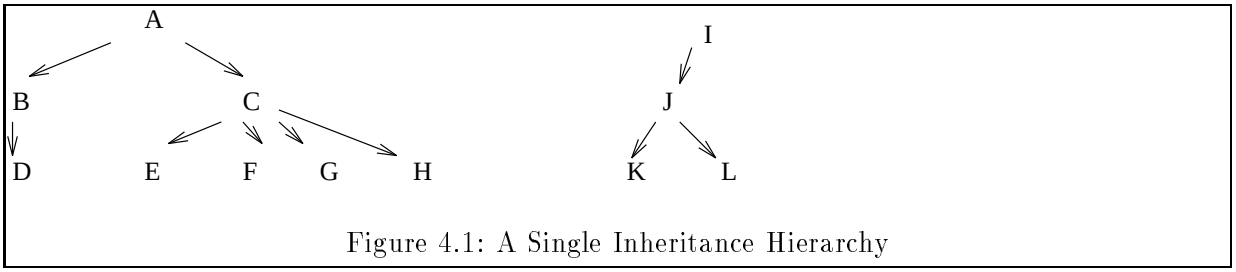


Figure 4.1: A Single Inheritance Hierarchy

In such models, inclusion polymorphism retains an important role. For example, a member of class `G` is also a member of classes `C` and `A`, and so can be treated as a `C` or an `A` when necessary. However, using single inheritance, it is not possible to represent the behaviour of a class which is a subclass of two different classes, which themselves are not related by a classification relationship. More precisely, in a single inheritance model, $(A \sqsubseteq B \text{ and } A \sqsubseteq C) \Leftrightarrow (B \sqsubseteq C \text{ or } C \sqsubseteq B)$. Often this restrictive view of inheritance is not desirable.

4.1.1.2 Multiple Inheritance

In multiple inheritance models, classes are not restricted in the number of parents they can have. The class hierarchy in figure 4.2 illustrates a typical multiple inheritance model.

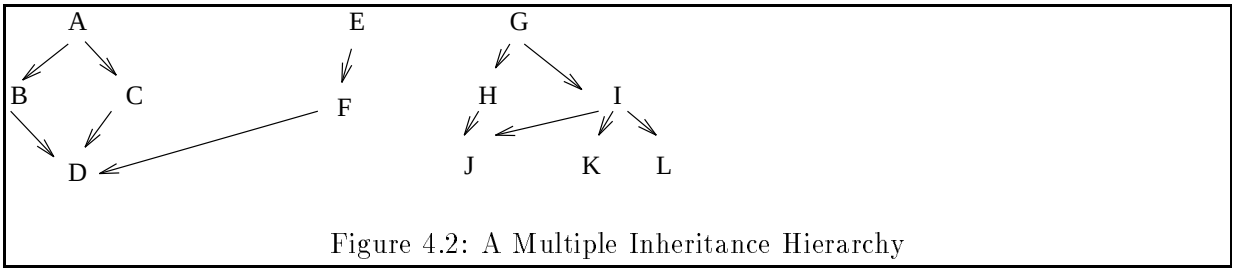


Figure 4.2: A Multiple Inheritance Hierarchy

Consider class `D`, in figure 4.2. Class `D` is a direct subclass of classes `B`,`C` and `F`. However, $C \not\sqsubseteq E$ and $E \not\sqsubseteq C$, for example. Class `D` offers the behaviour of both `C` and `F` even though these classes may not be related in any other way.

Multiple inheritance is such an important object oriented modelling concept that we must question why it is that some object oriented languages do not offer a multiple inheritance facility (see chapter 6). The answer is quite simply that in some cases, depending on the exact nature of the inheritance mechanism in the language in question, it is more trouble than it is worth. Fortunately, this is not the case with OO ACT ONE.

4.1.1.3 Multiple Inheritance is Problematic Only When Inheritance is not Subclassing

In object oriented programming languages, multiple inheritance can, and usually does, cause implementation difficulties. This is because these languages use inheritance as a code re-use facility rather than as a means of explicitly identifying behavioural compatibility. Object oriented programming languages make it difficult to distinguish between subclassing and composition. Using OO ACT ONE, the formal definition of these two relationships makes it much easier to distinguish between them. Multiple inheritance makes it appealing, when behaviour compatibility is not enforced, for programmers to use inheritance compositionally.

4.1.2 Inclusion Polymorphism and Dynamic Binding

The principle behind polymorphism is that a value (object) should not be constrained to being strictly typed (as a member of only one class). Polymorphic object oriented systems allow an object to be

treated as a member of more than one class. This type of property should set alarm bells ringing when it is first met: how are polymorphic systems type checked? Polymorphism seems to imply that no static type checking can be incorporated in an object oriented model which adheres to this flexible approach. This is true only for uncontrolled polymorphism.

In the most general case, uncontrolled polymorphism means that an object can be treated as if it is a member of any class. In other words, type checking is non-existent. Consequently, unless the programmer is very careful, errors can occur when an object is asked to provide a service which is not part of its external interface. Uncontrolled polymorphism is not a good feature for any language to exhibit.

Languages with hierarchical categorisation mechanisms are amenable to inclusion polymorphism. Two such types of language are those which include the notions of:

- types and subtyping.
- classes and subclassing.

In a typed language a value of one type can always be used in place of a value of any of its supertypes. In a classed language, an object of one class can always be used in place of an object in any of its superclasses. Inclusion polymorphism in typed languages can guarantee only the non introduction of syntax errors when a value of one type is replaced by the value of another. Inclusion polymorphism in a classed language can, and should, guarantee that a behavioural equivalence between a system before and after a class member in that system is replaced by the corresponding member in one of its subclasses.

The notion of substitutability is central to polymorphism. We must address the question of what it means to be able to accept a member of one class in place of a member of another class. It is difficult to address such a question without reference to a particular language. In OO ACT ONE there are two instances of substitutability:

- **Creating new objects from component parts**
  In OO ACT ONE, `STRUCTURE` operations are the means by which objects can be constructed from components. The parameter values of the `STRUCTURE` operations are the components of the new object being created. Inclusion polymorphism allows a new object to be created from component objects which are members of the class specified by the `STRUCTURE` operation defined, or members of a subclass of the class specified. For example, an object which is a pair of `integers` can also be created, using the same `STRUCTURE` operation, from a pair of `odd-integers` (provided `odd-integers` $\sqsubseteq$ `integers`).
- **Input parameters in external attributes**
  In OO ACT ONE, external attributes of a class can be parameterised on input values. A parameter value can be a member of the class specified in the operation definition, or a member of a subclass of the class specified. For example, an integer object with attribute `add <integer>` can be asked to add an `odd-integer` (again provided the appropriate subclassing relationship is explicitly defined in the OO ACT ONE code).

These two types of polymorphism are powerful mechanisms in an object oriented analysis model.

Dynamic binding is primarily an implementation concern. When an object is asked to service a request, the mechanism which it uses to service the request can be bound to the object at the time the request is placed. This is dynamic binding. It is not an analysis issue because in our object oriented requirements model we do not stipulate how the services between client and server should be provided.

4.1.3 OO ACT ONE: An Explicit Subclassing Approach

OO ACT ONE supports the specification of multiple inheritance hierarchies in a peculiar way. When one class, A say, is identified as being a subclass of two other classes, B and C say, there are two ways, in general, for this behaviour to be specified:

- i) First specify B, then explicitly define A to be a subclass of B and, finally, specify C as a superclass of A.
- ii) First specify A, then explicitly define B to be a superclass of A and, finally, specify C as a superclass of A.

In each of these specifications the resulting class hierarchy is the same.

Note that multiple inheritance is possible in OO ACT ONE because we provide explicit superclassing mechanisms, defined as inverses of the explicit subclassing mechanisms. In object oriented programming all explicit class relationships (like inheritance) allow a new class to be defined only as a subclass of an already existing class. OO ACT ONE allows the relationship to be defined in the other direction: a new class can be defined to be a superclass of an already existing class. This is a new approach to the definition of object oriented behaviour which facilitates multiple inheritance modelling in OO ACT ONE.

4.1.4 Abstract Classes

The term *abstract superclass* pervades object oriented programming languages. It arises in the following type of scenario:

> A class of shapes (for display on a screen, say) is a superclass of triangles, squares and pentagons. Each of these subclasses exhibits the behaviour of superclass shapes. It is not possible to instantiate a member of the shapes class which is not a triangle, square or pentagon. The shapes class is an abstract superclass.

This notion is more formally represented in OO ACT ONE by a class which is defined to generalise another class, or classes, without defining new literal or structure operations. In terms of the O-LSTS semantics, the definition of an **abstract superclass** is given below.

Definition: Abstract Superclass

> Class C is an **abstract superclass** in $Env_D \Leftrightarrow$
> $\forall c \in US(C), \exists C' \in visible(D)$ such that $c \in US(C')$ and $C' \sqsubseteq C$ in Env_D.

4.1.5 A Polymorphism Problem: Heterogeneous Data Stores

Consider a class of `item`s, which is an abstract superclass with subclasses `integer`, `character` and `bool`. We wish to define a data store (a `stack` say) of `item`s with external attributes `push` and `pop`. The main difficulty with this type of behaviour is the unidirectional aspect of polymorphism in object oriented languages: an object can be polymorphised 'up the class hierarchy but not down it'. In other words, a member of a class C can be treated as a member of a superclass of C but not a member of a subclass of C. Consequently, for example, if an `integer` *becomes* an `item`, when it is `push`ed onto an `item stack`, then when it is `pop`ped off the stack it remains a member of the `item` class and cannot be used as an `integer`.

The O-LSTS semantics conforms to the unidirectional polymorphic model. There are two reasons for defining the semantics in this way:

- It simplifies the understanding of O-LSTS behaviour since it is always clear how each object is classified.
- It simplifies the semantics since it is not necessary for every object to remember its polymorphic history. For example, an `item` does not need to know that it was once an `integer` which was once an `odd-integer` ....

4.2 Composition

The composition relationship is fundamental to object oriented analysis and requirements capture. Human understanding of systems is based on a divide-and-conquer deconstructionist approach, which can be summarised as follows:

> If we wish to understand an object A then by identifying the parts A is made from, and attempting to understand these, the original problem of understanding A is simplified.

There are three questions which we need to ask about this deconstructionist philosophy:

- 1) Is there always a unique set of components associated with a system being analysed?
- 2) Why should a component of a system be easier to understand than the system in which it is found?
- 3) How do we know when the deconstruction of understanding should end?

The answer to the first question is certainly *NO*, otherwise analysis would be 'trivial'. The second question is less easy clear since the rather paradoxical answer is that sometimes components of a system are more difficult to understand than the system in which they are found. This is especially true when a component is used, in the particular system being analysed, to provide only a small set of the behaviours which it is capable of exhibiting. In such a system it may be easy to understand the limited behaviour of such a component. However, if such a component is separately analysed then the usage constraints are no longer relevant and understanding its complete behaviour becomes very difficult. (This argument illustrates quite strongly why abstraction is such a powerful mechanism

for human understanding. Deconstructionism depends on the ability to be able to identify useful composition abstractions.) The final question is the bane of many an analysis process: too many systems are over-analysed leading to an increase in development costs and bad design. It is vital that an analysis method incorporates a means of deciding when the requirements capture process is complete.

4.2.1 Composition Structure

With respect to analysis alone it has already been stated that there are many ways of structuring problem understanding. An analyst must chose the structure best suited to communicating the requirements model with the customer. The designers should not be considered at this stage: they should be familiar with the object oriented analysis models and therefore be able to cope with any structure defined within the figure=FormAnal-Chp4/Figures semantic framework.

The compositional structure is fundamental to object oriented analysis. OO ACT ONE has an explicit mechanism for capturing structure properties: the `STRUCTURE` operation. This makes composition one of the most visible aspects of an OO ACT ONE specification.

Composition is a relationship between a container object and its contained parts. It tells us nothing about the relationship between the parts. Configuration and interaction are two relationships which arise out of one object being decomposed into distinct parts. Two objects may configure and interact only if they are components of a common container.

4.2.2 Configuration

`STRUCTURE` operations only brush the surface with respect to the composition analysis of problems. One can state that a car is composed from an engine, a chassis, a suspension, body, wheels, lights, etc ...(and each of these components can themselves be further decomposed) but this does say how the components are connected together (or if they are connected at all).

In section 3.3.5, configuration of components is formally defined as an interdependency during the fulfilment of a service request. Informally, two components configure if at least one of the services provided by their containing object depends on both the components to fulfil that service. This is a very abstract way of conceptualising configuration, but one which is appropriate to analysis. Analysis identifies *what* not *how*.

4.2.3 Interaction (Data Flow and Control Flow)

Interactions, with associated data and control flow, are much more concrete notions than configuration. They describe *how* behaviour is fulfilled rather than only saying *what* the behaviour requirements are. These notions are common to structured analysis methods but are not given prominence in the object oriented analysis advocated in this thesis: they are considered in greater detail as part of the object oriented design process. An interpetation of these concepts can be derived from an OO ACT ONE specification, but such an interpretation should be taken only in the following two circumstances:

- When the customer is more familiar with structured analysis methods than with object oriented techniques.
- When designers require a more traditional interpretation of a requirements model.

Appendix D details a formal interpretation of interaction, data-flow and control-flow in OO ACT ONE specifications.

4.2.4 Structures: Dynamic and Static

The notions of dynamic and static structure are important when attempting to provide an interpretation of composition properties. This section shows the difficulties that dynamic structures give rise to in an object oriented analysis model. They are open to abuse in the sense that, if used *wrongly*, they can make specifications hard to communicate with the customer. A *pure* style of specification is introduced to overcome this potential problem.

4.2.4.1 *Impure* Object Oriented Specification Practices

Impure Style: Example 1

The following OO ACT ONE specification of class `TwoStacks` is well defined but illustrates an *impure* style of specification.

```
CLASS TwoStacks USING Stack, Nat OPNS
STRUCTURES: S<Stack, Stack>
DUALS: pop -> Nat
TRANSFORMERS: push<Nat>, swap
EQNS
S(Stack1, Stack2).pop = S(Stack1, Stack2.pop) AND Stack2..pop;
S(Stack1, Stack2).push(Nat1) = S(Stack1.push(Nat1), Stack2.pop)
S(Stack1, Stack2).swap = S(Stack2, Stack1)
ENDCLASS (* TwoStacks *)
```

The behaviour specified in `TwoStacks` is straightforward to understand but on closer examination raises some interesting points for designers. For example, the `Stack` components may be implemented on different processors at different sites and conceptually the specification seems to suggest that a `swap` service results in all the data stored in one `Stack` being transferred to the other (and vice versa). Of course, this is an inefficient way of implementing such behaviour. It is more natural to define a pointer component which addresses either `Stack1` or `Stack2` and makes the state of the pointer change in response to a `swap` request. Such a behaviour is specified below in class `TwoStacksB`. (In O-LSTS semantics it is simple to show that these specifications fulfil each others behaviour.)

The `TwoStacksB` specification is written in a *pure* object oriented style — the components of a `TwoStacksB` object persist throughout the life-time of the object and are used only through their external interfaces. In the first class specification of `TwoStacks` the structure components do not persist, even though the structure is fixed.

Impure Style: Example 2

```
CLASS TwoStacksB USING Stack, Nat, Bool OPNS
STRUCTURES: S<Stack, Stack, Bool>
DUALS: pop -> Nat
TRANSFORMERS: push<Nat>, swap
EQNS
Bool1 => S(Stack1, Stack2, Bool1).pop = S(Stack1, Stack2.pop, Bool1) AND Stack2..pop
OTHERWISE S(Stack1.pop, Stack2, Bool1) AND Stack1..pop;
Bool1 => S(Stack1, Stack2, Bool1).push(Nat1) =
S(Stack1.push(Nat1), Stack2.pop, Bool1)
OTHERWISE S(Stack1.push(Nat1), Stack2.pop, Bool1);
S(Stack1, Stack2, Bool1).swap = S(Stack1, Stack2, Bool1..not)
ENDCLASS (* TwoStacksB *)
```

```
CLASS StacksAgain USING Stack, Nat OPNS
STRUCTURES: S-SA< Stack, Stack>
DUALS: pop -> Nat
TRANSFORMERS: push<Nat>, move
EQNS
S-SA(Stack1, Stack2).pop = S-SA(Stack1.pop, Stack2) AND Stack2..pop;
S-SA(Stack1, Stack2).push(Nat1) = S-SA(Stack1.push(Nat1), Stack2);
S-SA(Stack1, Stack2).move = S-SA(Stack1.pop, Stack2.push(Stack1..pop))
ENDCLASS (* StacksAgain *)
```

This `StacksAgain` specification is also well defined but written in an *impure* style. The `pop` operation is defined *impurely* — it uses the result of a dual operation on the second component without changing its state accordingly. In a *pure* object oriented style it is required that when dual operations are requested the accessor part of the attribute cannot be utilised without the transformer acting on the internal state of the serving object. In other words, although OO ACT ONE conceptually allows the two parts of a dual to be used separately, a *pure* specification style does not let an accessor part of a dual to appear in the right hand side of an equation definition without the new state of the serving component being updated to model the servicing of the whole dual. Note that such a check only makes sense in a class which is defined to have persistent components.

4.2.4.2 Object Oriented Interpretation of Dynamic Structure

It is not possible to extend these pure styling conventions to cope with objects which have dynamic structure. In particular, objects defined using recursive structure operations are difficult to reason about compositionally. For example an integer stack object is illustrated using an object structure diagram in figure 4.3.

In the same figure a new representation is introduced for more concise diagrammatic representation of recursive structures. The formal meaning of both diagrams is given by the OO ACT ONE `STRUCTURE` expression which is common to both.

The syntax of the new diagram is representative of the way in which linked list structures (common to all forms of programming) have an informal yet powerful means of representation (as nodes

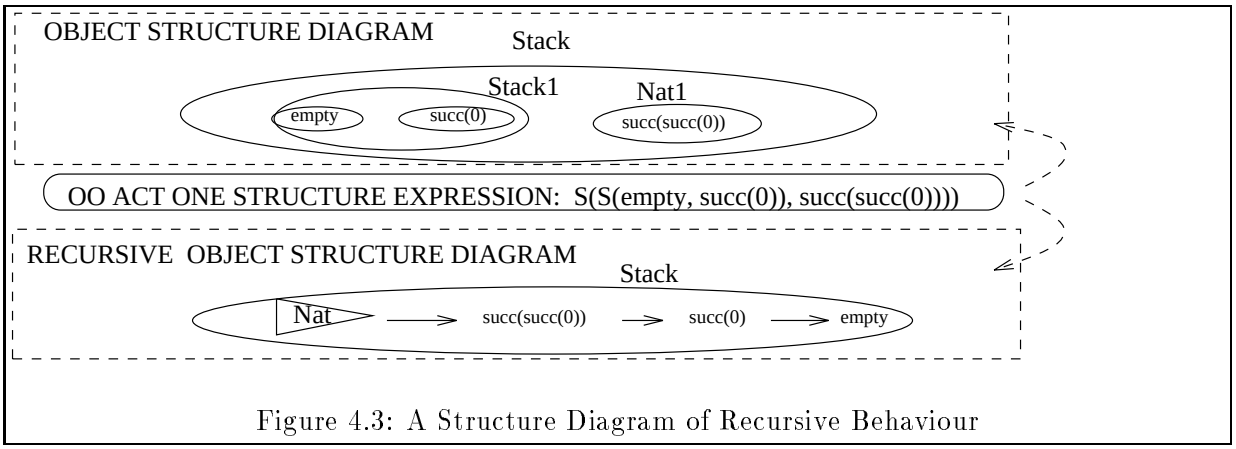


Figure 4.3: A Structure Diagram of Recursive Behaviour

and pointers). The recursive structure diagram emphasises that, although objects with recursive structure have a complex embedded structure, the consistent relationship between components and subcomponents (and subcomponents and their subsubcomponents, etc ...) means that such objects are often understood in a linear or tree like fashion.

Non-recursive Dynamic Structures

A non-recursive dynamic structure is specified in OO ACT ONE when the structure expression on the left hand side of an equation does not match the structure expression on the right hand side of the equation. This models an object (or objects) in a class which change their internal structural composition in response to a service request. Modelling such behaviour is very powerful but should be used only to represent special events in the life of an object. It should not be the normal means of defining behaviour, since if it is not done sparingly it can severly reduce the clarity of the intended meaning.

4.2.5 Shared Objects

In OO ACT ONE there is no notion of one object being shared between others. This thesis argues that such an idea is implementation oriented. It is not desirable for an analyst to worry about such things. A simple example, the `TwinFunction` class, illustrates this quite clearly.

The class structure diagram for this specification is given in figure 4.4.

In this example, a reasonable interpretation is that the two function components share the two queues[1]. For example, both function components **depend on** `Queue1`, and so `Queue1` can be implemented as a (persistent) object shared between `Function1` and `Function2`. This sharing interpretation is not an analysis issue. A designer could choose a solution architecture in which sharing is not evident.

[1] An alternative, but equally valid interpretation, is to say that the two queues share the two functions.

```
CLASS TwinFunction USING Function, Queue, Nat OPNS
STRUCTURES: S-TF<Queue, Queue, Function, Function>
DUALS: remove -> Nat
TRANSFORMERS: add<Nat>, process1, process2
EQNS
S-TF(Queue1,Queue2,Function1,Function2).remove =
S-TF(Queue1,Queue2.pop,Function1,Function2) AND Queue2..pop;
S-TF(Queue1, Queue2, Function1, Function2).add(Nat1) =
S-TF(Queue1.push(Nat1), Queue2, Function1, Function2);
S-TF(Queue1, Queue2, Function1, Function2).process1 =
S-TF(Queue1.pop, Queue2.push(Function1.in(Queue1..pop)), Function1, Function2);
S-TF(Queue1, Queue2, Function1, Function2).process2 =
S-TF(Queue1.push(Function2.in(Queue2...pop)), Queue2.pop, Function1, Function2)
ENDCLASS (* TwinFunction *)
```

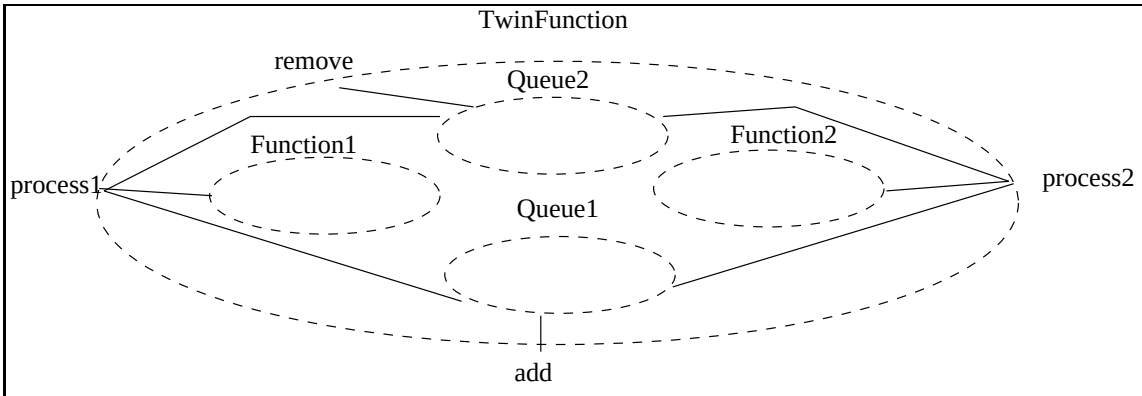


Figure 4.4: Sharing is not an Analysis Issue: An Example

4.2.6 Timing Properties

The formal specification of timing properties has been the basis of a wide range of research in recent years. Some semantic models are intended for general use (see [63], for example) whilst other semantics have been given for more specific formal models: LOTOS, for example, has a simple timing model for defining behaviour as a temporal ordering of events. There has been much work done to extend the LOTOS timing model (see [11], for example). It is useful to ask what sort of timing properties a formal object oriented requirements model should be able to exhibit, and whether OO ACT ONE is suitable for specifying these properties.

4.2.6.1 A Categorisation of Timing Properties

For discussion purposes, timing properties are categorised into four groups:

- **Event Sequencing**
  LOTOS exhibits such a timing model. The sort of behaviour that this model specifies is 'event A must occur before event B can occur'. In object oriented analysis, events correspond to service requests at the external interface of an object. Now, our object oriented semantics state that all

service requests in the external interface of an object can be serviced at any time during that object's lifetime. Consequently, external event sequencing properties are not relevant. Section 4.2.6.2 formulates an interpretation of internal event sequencing properties for structured objects specified in OO ACT ONE.

- **State Sequencing**
  State sequencing is logically equivalent to event sequencing. The sort of behaviour specified in state sequencing models is 'object A cannot be in state S before it has been in state T'. Such timing properties can be taken directly from the O-LSTS semantics for OO ACT ONE specifications. However, such behaviour properties are not explicit in the OO ACT ONE and can only be deduced after some appropriate analysis.
- **Synchronisation**
  One of the most common timing properties that is used in the specification of behaviour is synchronisation. For example, 'Object A must synchronise with object B on event C'. Other than for communication purposes, synchronisation constraints are predominantly design and implementation issues. It is beyond the scope of this work to extend the O-LSTS semantics to facilitate the explicit statement of such properties during object oriented analysis.
- **Quantitative Timing**
  We are not concerned in our object oriented semantic model with being able to specify timing properties with respect to some sort of global passing of time.

4.2.6.2 Interpretation of Timing Properties in an OO ACT ONE Specification

Two interpretations of internal timing properties are given for OO ACT ONE specifications of structured objects:

I) Ordering of Internal Service Requests

A static analysis of OO ACT ONE equation definitions is sufficient to identify 'timing properties' which are implicit in an object oriented requirements specification. For example, in `TwinFunction` (see 4.2.6) the equation:

```
S-TF(Queue1, Queue2, Function1, Function2).process1 =
S-TF(Queue1.pop, Queue2.push(Function1.in(Queue1..pop)), Function1, Function2);
```

can be given the following interpretation:

> When servicing a `process1` request, the `Queue1` component must have a `pop` serviced not after the `Function1` component has an `in` serviced, which in turn must then occur not after the `Queue2` component has a `push` serviced.

In an implementation, the component services cannot occur concurrently, since the result of one service is used as input to another. Therefore the phrase 'not after' can be read as 'before', when design and implementation issues arise. Designers can interpret the specification as saying that the internal `pop` occurs before the internal `in` which occurs before the internal `push`, which all result from one external `process1` request. This property is not an explicit part of the object oriented semantics

which do not specify how internal state transitions are achieved.

II) Synchronisation of Internal Requests

Consider a system of two stacks and one queue component. Informally, the behaviour of such a system is defined as follows. Natural numbers are pushed onto the queue. They can be moved synchronously to both the stacks and separately popped off either stack. This is precisely stated in the OO ACT ONE specification of class `QSS`.

```
CLASS QSS USING Queue, Stack, Nat OPNS
STRUCTURES: S-QSS<Queue, Stack, Stack>
DUALS: pop1 -> Nat, pop2 -> Nat
TRANSFORMERS: push<Nat>, move
EQNS S-QSS(Queue1, Stack1, Stack2).push(Nat) = S-QSS(Queue1.push(Nat), Stack1, Stack2);
S-QSS(Queue1, Stack1, Stack2).pop1 = S-QSS(Queue1, Stack1.pop, Stack2) AND Stack1..pop;
S-QSS(Queue1, Stack1, Stack2).pop2 = S-QSS(Queue1, Stack1, Stack2.pop) AND Stack2..pop;
S-QSS(Queue1, Stack1, Stack2).move =
S-QSS(Queue1.pop, Stack1.push(Queue1..pop), Stack2.push(Queue1..pop))
ENDCLASS (* QSS *)
```

The 'synchronisation' of the two `Stacks` on the `move` operation is represented by the **state label expression** `Queue1..pop` appearing twice in the **state label expression** on the right hand side of the `move` equation definition. The result of `Queue1` popping off a value is used by both the `Stacks`. In the OO ACT ONE specification there is no explicit statement that both `Stacks` synchronise. This synchronisation is one way the designer of a solution to the system can guarantee that the same value is given to both components: it is not the only solution.

4.2.6.3 Timing is a Design Concern

This section has shown that timing is not directly an analysis concern. It is necessary that an object oriented requirements model can be interpreted by designers and so an informal mechanism for extracting timing properties with respect to internal interaction between peer components has been formulated. However, OO ACT ONE does not contain explicit timing mechanisms. The timing aspects of an object oriented requirements model are deliberately abstracted away from.

4.3 Other Object Oriented Analysis Issues

4.3.1 Concurrency

In OO ACT ONE, the servicing of a request is defined as the evaluation of a **state label expression**. The subexpressions of a **state label expressions** may be evaluated independently, and this can be given a concurrent interpretation. Concurrency is examined in more detail in section 6.5.

4.3.2 Communication: Synchronous vs Asynchronous

All computing systems exhibit communication properties. There are fundamentally two different communication models: synchronous and synchronous. During object oriented analysis and requirements capture, it is important that the communication model is abstracted away from.

4.3.3 Exception Handling

`Unspecified` class members are used to model exceptions. When behaviour is required which cannot be modelled in the requirements without making implementation decisions then the analyst can chose to model this behaviour using an `unspecified` value. It is important that these exceptions are not dealt with prematurely by the analyst. The `unspecified` mechanism allows the analyst to abstract away from *how* exceptions are implemented to identify only *what* exceptions must be considered at later stages of development.

4.3.4 Nondeterminism and Probabilistic Behaviour

Nondeterminism is a powerful specification facility. Until now, the systems we have analysed have all been deterministic. The reason for ignoring nondeterministic behaviour until this stage is that it is not necessary to change the semantics to record this type of behaviour. Nondeterminism is recorded by commenting `TRANSFORMER` operations as being `INTERNAL`. An `INTERNAL` operation need not be initiated through the external interface of the object in question.

In the ACT ONE executable model, the `INTERNAL` transitions are treated no different from the others. However, in a concurrent model of the requirements, the `INTERNAL` transitions can occur independent of the environment in which the object in question is found. The commenting of transitions as `INTERNAL` keeps the semantics simple whilst ensuring that the designers are explicitly informed of the nondeterminism.

Nondeterminism is used in two ways:

- To model probabilistic behaviour.
- To specify implementation freedom.

Both these aspects are important in the analysis of a system.

4.3.4.1 Probabilistic Behaviour

Consider the `CoinToss` class, defined below.

```
CLASS CoinToss USING Bool OPNS
STRUCTURES: Coin<Bool>
ACCESSORS: Toss -> Bool
TRANSFORMERS: HorT<Bool> (*INTERNAL*)
EQNS CoinToss.HorT(Bool1) = Coin(Bool1); Coin(Bool1)..Toss = Bool1
ENDCLASS (* CoinToss *)
```

A valid implementation of this OO ACT ONE class can, for example: always respond `true` in response to a `Toss` request, or always respond `false`, or alternate between `true` and `false` responses (to name but three options). When nondeterminism is used to model random behaviour then the analyst must record the probabilisitic requirements outside the OO ACT ONE framework of specification. (It is beyond the scope of this thesis to examine how this can be done: [63] gives one particular view of probabilistic semantics which may be useful in this respect.)

4.3.4.2 Implementation Independence

Consider the behaviour specified by class `SysQSS`, below.

```
CLASS SysQSS USING Queue, Stack, Nat OPNS
STRUCTURES: SystemQSS<Queue, Stack, Stack, Bool> DUALS: pop1 -> Nat, pop2 -> Nat
TRANSFORMERS: push<Nat>, move, PickStack<Bool> (*INTERNAL*)
EQNS SystemQSS(Queue1,Stack1,Stack2,Bool1).push(Nat1) =
SystemQSS(Queue1.push(Nat1),Stack1,Stack2,Bool1);
SystemQSS(Queue1, Stack1, Stack2, Bool1).pop1 = SystemQSS(Queue1, Stack1.pop, Stack2, Bool1) AND
Stack1..pop;
SystemQSS(Queue1, Stack1, Stack2, Bool1).pop2 = SystemQSS(Queue1, Stack1, Stack2.pop, Bool1) AND
Stack2..pop;
Bool1 => SystemQSS(Queue1, Stack1, Stack2, Bool1).move = SystemQSS(Queue1.pop,
Stack1.push(Queue1..pop), Stack2, Bool1)
OTHERWISE SystemQSS(Queue1.pop, Stack1, Stack2.push(Queue1..pop), Bool1);
SystemQSS(Queue1,Stack1,Stack2,Bool1).PickStack<Bool2> = SystemQSS(Queue1,Stack1,Stack2,Bool2)
ENDCLASS (* SysQSS *)
```

This class models implementation freedom: when a `move` request is serviced, the implementer is free to decide how the system chooses which `Stack` component the `Queue` should transfer its data to.

4.3.5 Active and Passive Objects

In most object oriented systems it is common to distinguish between active and passive objects:

- An active object is likened to a process whose existence persists over a sequence of events.
- A passive object is likened to a piece of data which flows between active objects.

In this thesis, this potentially confusing distinction is not made: the active and passive concepts seem to have arisen from implementers attempting to conceptualise objects in non-object oriented terms.

When using OO ACT ONE, there is a more useful division than active and passive: static and dynamic. Static objects are those whose classes do not offer transformer attributes. Dynamic objects are those whose classes do offer transformer attributes. Static objects are important because they can be implemented as shared objects, without risk of the principle of encapsulation being broken.

4.3.6 Persistency

When an object oriented system is created, a component of the system is said to *persist* if it is identifiable throughout the lifetime of the system. When a transformer request is serviced, the new state of an object may be constructed from the same components. This interpretation can be taken when the OO ACT ONE specification is defined *purely* (see 4.2.4.2).

Dynamic objects which are *purely* defined can be implemented as a fixed set of components. History dependent behaviour is then realised by the set of persistent component objects changing their internal state through services delegated to them (by their containing object) at their external interfaces.

The notion of persistency is not explicit in the OO ACT ONE requirements model. We recommend that a need for persistency should be recorded in the informal parts of the requirements document. Then, a simple static analysis of the OO ACT ONE code can check to ensure that the class in question is *purely* defined. It is not an error if the specification is *impure*, but a warning message can be given to state that persistency cannot be guaranteed.

4.3.7 Class Routines: Configuration and Creation

Creation and configuration routines provide a means of restricting the set of initial states which an object of a specified class can attain. In OO ACT ONE, an `(* INITIAL *)` comment can be used to identify the literals and structures (and possible invariants on the structures) which can be used to define a newly created object. Such a comment can be used to stimulate a static analysis to check that all dynamically created objects are correctly initialised. (Such an analysis is beyond the scope of this thesis.)

4.4 Reviewing the OO ACT ONE Specification Language

4.4.1 Does It Meet Our Expressional Requirements?

The fundamental requirements for an object oriented analysis language are that it provides a means of recording relevant information which: increases problem understanding, is amenable to customer validation, and results in a formal requirements model useful to designers. OO ACT ONE fulfils all these requirements.

In section 2.2.3 we identified 'features of good analysis methods' as a list of criteria by which analysis methods can be judged. This section re-examines these criteria with respect to OO ACT ONE.

- **i) Amenability to change within a stable structure**
  The classes, objects and object compositions, within a system, provide a stable base upon which changes to requirements can be easily made.
- **ii) Encouragement of Re-Use**
  Three types of re-use are evident in OO ACT ONE:

  - **Compositional** — re-use of predefined components and structures.
  - **Categorisational** — re-use of behavioural characteristics as defined in a class hierarchy.
  - **Experience** — OO ACT ONE is simple and straightforward since it is built around a small set of well understood concepts and precisely defined mechanisms. Consequently, this thesis argues that analysts can quickly gain experience in developing technique founded on experience, rather than learning the underlying model, when coding in OO ACT ONE.

- **iii) Interfacing between customers and designers**
  The 5-model approach to specification is prominent in OO ACT ONE. This approach is *customer oriented* in the sense that it arose from a model of customer understanding and is amenable to customer validation. The diagrammatic notations are accessible to customers, analysts and designers alike, whilst the underlying mathematical model provides the formal basis upon which one coherent framework of understanding is built.
- **iv) Incorporating Standard Modelling Techniques**
  We consider five standard modelling practices to be inherent in OO ACT ONE specifications:

  - Abstraction (and encapsulation) are fundamental to the object oriented model.
  - (De)composition is provided by `STRUCTURE` operations
  - Class hierarchy constructs are provided by explicit subclassing mechanisms.
  - Communication between objects is modelled as one object requesting a service of another. The service is requested so that the **client** can utilise the encapsulated behaviour offered by the **server**. The information that flows between the objects is defined by the input and output parameters.
  - Model co-ordination is prominent in an OO ACT ONE specification, where there are many different views of the behaviour defined, which combine in a consistent and coherent fashion.

- **v) Having a formal basis**
  The OO ACT ONE language is rigorously defined using the O-LSTS semantics.

4.4.2 Is OO ACT ONE Purely an Analysis Language?

Sections 4.2 and 4.3 emphasise the number of object oriented issues which are abstracted away from when using OO ACT ONE during analysis and requirements capture. The most design-like features of OO ACT ONE specifications are the `STRUCTURE` operations. The composition relationship is a characteristic which arises from problem domain structure, rather than from some arbitrary structure imposed by the analysts to aid their understanding. This thesis argues that it is necessary for such structure to be prominent in a set of object oriented requirements.

4.5 The Practicalities of Building a Formal Model

There are three aspects of analysis which group together under the heading 'practicalities':

- **Models:** syntax and semantics.
- **Development Method:** how to generate, test and change the models to achieve the best requirements model.
- **Tools:** automated mechanisms for support of model building, execution, validation and verification

Chapters 2 and 3 have developed a number of different analysis models. Section 4.5.1 defines a method[2] (called the *skeleton* approach) for the development of OO ACT ONE requirements models.

4.5.1 The Skeleton Method to Object Oriented Analysis

The whole *skeleton* strategy is based around the application of seven different processes, each of which can be applied at any time during the analysis and, with careful control, can often be applied in parallel. These processes are shown in figure 4.5.

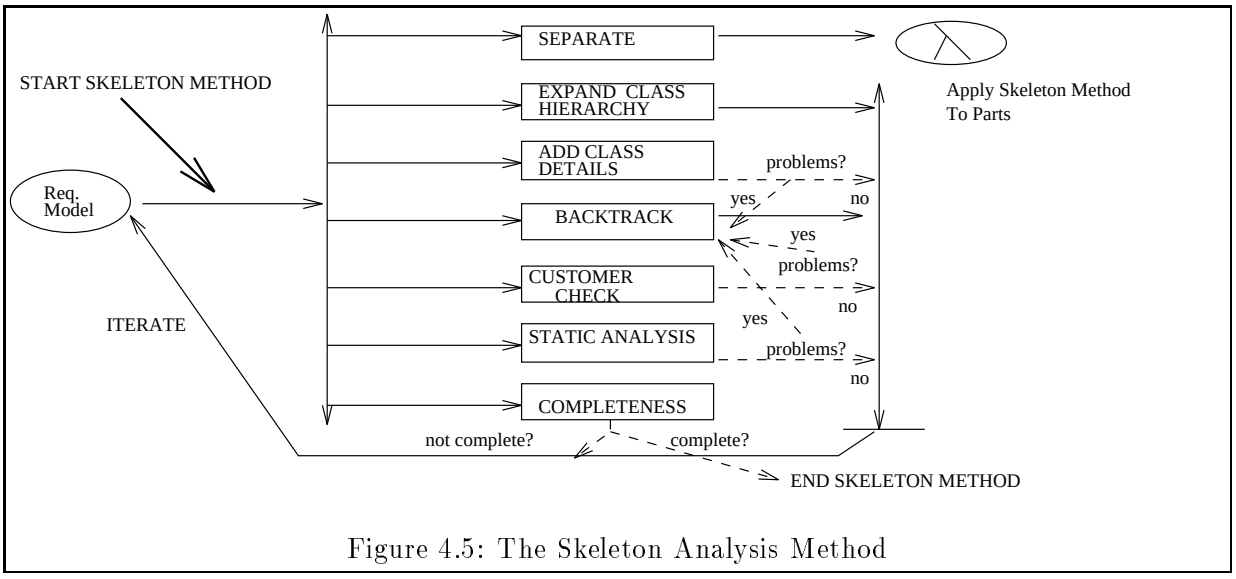


Figure 4.5: The Skeleton Analysis Method

4.5.1.1 The Opportunistic Algorithm

The following simple algorithm provides a framework upon which a complex analysis method can evolve.

BEGIN Skeleton Method
REPEAT
Choose a process (or processes) from **(i) to (vii)**

- **(i) Separation**
  Recognise distinct and separate parts of the system and apply the skeleton approach to each of these. Reconnect the parts after their completion.

[2]The method is no more than a simple algorithm which can be followed when learning how to use OO ACT ONE. A proper analysis method should evolve from this very primitive starting point.

- **(ii) Expand Class Hierarchy**: either
  - Identify a new class in the system and add to hierarchy without defining it as a subclass or superclass of any other classes in the hierarchy.
  - Identify a new class explicitly as a (sub/super)class of an already identified class.
  - Change a class which is separate from the other classes by explicitly defining it as a (super/sub)class of an already identified class.
- **(iii) Add Details to some class in the hierarchy**: either
  - Identify literal or structure members.
  - Identify invariant properties.
  - Specify an external attribute of a class as an accessor, transformer or dual operation.
  - Specify the behaviour associated with an external attribute in an equation definition (provided it has not been previously defined).
- **(iv) Backtrack**: either
  - Disconnect a (sub/super)class link in the hierarchy.
  - Delete a class which is no longer needed (perhaps because it has been superseded by another).
  - For some class in the hierarchy, remove literal or structure operations.
  - Delete an external attribute.
  - Delete an equation associated with an external attribute.
- **(v) Check with the customer**: either
  - Execute, using the ACT ONE dynamic model, one or more of the classes whose behaviour is complete. In other words, validate customer expectations of the dynamic behaviour of the system (or a component of the system) being analysed.
  - Present the customer with graphical views of the OO ACT ONE code: class hierarchies, structure diagrams and O-LSTSD(s). This helps to check analyst understanding against customer understanding of the problem domain.

  Then, if the checks have identified 'problems' these must be noted and resolved (either by **backtracking** or by adding the extra understanding gained from customer interaction). Otherwise, the classes checked are noted as being validated and marked as such in the class hierarchy.
- **(vi) Statically Analyse Requirements Specification**
  The whole of the system or distinct parts (classes) can be checked for correctness using the static analysis defined in section 3.4. After a successful analysis, the ACT ONE execution model can be used by the analyst alone to test their understanding of the specification. Problems identified during the static analysis stage or the execution stage must be noted and corrected in the model.
- **(vii) Check The Analysis For Completion**
  The following tests must be made before the analysis can be declared complete:
  - Check that all customer classes (i.e. those classes identified by the customer in the problem domain) have been specified in the class hierarchy.
  - Statically Analyse the whole system (i.e. all the separate class trees) for correctness properties.
  - Validate that the customer is happy with the dynamic behaviour exhibited by every class in the specification (this should be done as the specification is developed so that the final check is straightforward) and the relationships between them. In particular check the classification, composition and configuration properties.

UNTIL analysis is checked as complete
END Skeleton Method

It is not practical for every single change to the analysis model to be formally tested, or validated by the customer, as they are made. It is up to the analysis manager to decide the *best* strategy for making intermediate checks.

During application of the *skeleton* approach, different parts of the system will be better understood and more completely specified than others. As analysis proceeds a skeleton class hierarchy gradually appears and this skeleton is held together by the classification and composition relationships. The meat on the bones is provided by the operation and equation definitions in each class.

Central to the skeleton method is some means of guaranteeing that one class of behaviour is not specified many times as a result of splitting the system up into separate analysis parts. Further, there must be some standard way of re-using already existing components, and for making newly defined component classes available for re-use.

4.5.1.2 Re-use and Costing

Project managers must take into account the need for different costing strategies when re-use is prominent in a development method. In an ideal environment one could pay for predefined components with the formal specification acting as some sort of contract between vender and consumer. However, it is more likely that companies will develop their own libraries of classes for use in-house. Software re-use is appealing in principle but does lead to many difficult questions in practice. It is beyond the scope of this thesis to examine the consequences of re-use in the environment in which the development is taking place.

4.5.1.3 Re-use: A Note On Libraries

In a proper object oriented development environment pre-defined classes of behaviour should be as easy to re-use as the application of any other language mechanism. The library classes should be looked upon as part of the semantics of the language. Object oriented programming languages are constructed from three parts:

- The language primitives.
- The re-use mechanisms.
- The libraries of re-usable components.

It is these three things together which combine to produce an object oriented language. Most object oriented languages come already with library class hierarchies to provide behaviour comparable to that provided as language primitives in larger imperative languages. Object orientation places more responsibility on the user to control the library facilities. Unfortunately, OO ACT ONE does not presently have a large set of library classes.

4.5.2 Validation

Other than the static analysis of an object oriented specification the only other means of testing the requirements model is by running event sequences on the ACT ONE executable model. The event traces produced (in response to a sequence of evaluations) must be validated against customer and/or analyst understanding of the requirements. In the execution of some behaviour three types of result occur:

- 1) The ACT ONE expression is evaluated (according to the well-defined re-write rules) to represent a value which has a valid object oriented interpretation:
  - When the operation is a transformer, the expression evaluates to a member of the **server class** represented as a literal a structure expression of the **server class** sort.
  - When the operation is a dual, or an accessor, the expression evaluates to a dual expression of the **server class**
- 2) The ACT ONE expression evaluates to an **unspecified** literal value of the appropriate class. This corresponds to behaviour which must be determined at a later stage in the development.
- 3) The expression does not evaluate to either of the cases above. This occurs only when the dynamic behaviour of an object results in an invariant property being broken. It is not possible to guarantee that all invariants are upheld during the lifetime of a class without severly restricting their use. For example, invariants could be permitted only in static classes (i.e those without transformer attributes). However, this restriction limits the power of expression in OO ACT ONE. It is up to the analysts to test the correctness of their specifications with respect to invariants. Consequently, the static analysis of OO ACT ONE warns the analyst when an invariant is used in a dynamic class. Analysts must prove that invariants are never broken otherwise correct behaviour cannot be guaranteed.

There are many different ways, in theory, that invariant properties can be guaranteed. This section identifies two solutions that work in practice.

- **Solution 1: Test Initial States and Validate Transitions**
  Given a list of initial states that an object in a particular class can be initialised to, it is necessary to first check that these states fulfil the invariants. Once this is done, it is necessary only to check that all invariant properties are upheld across the state transitions defined in the transformer and dual equations.
- **Solution 2: Ignore Transformers Which Result In Broken Invariant Properties**
  All transformer operations are defined to result in no change of state when an invariant would be broken as a consequence of their being fulfilled in the normal way. Dual operations are similarly defined, with the additional property that the unspecified value of the appropriate class is returned as the result of an operation which results in a broken invariant. This convention guarantees that the state of an object always fulfils the invariants defined on it.

There are certainly *better* approaches to dealing with invariant validation but this is a general problem which was not examined in this thesis.

4.5.3 Tools

This thesis has presented a set of models for developing object oriented requirements specifications and a method for utilising these models. However, the thesis does not present any tools for aiding this process. Clearly, tool support is vital to all development processes. Rather than creating these tools, this thesis identifies the type of tools which can and should be developed as future work:

- A customer friendly, animation tool, is required. This tool can be supported by the ACT ONE evaluation mechanism.
- A tool to aid the analyst in the synthesis of OO ACT ONE specifications is required. In particular, some sort of library browsing facility is required whereby components can be cut and pasted in and out of a system hierarchy.
- There is a need for tool support in the areas of static analysis and verification.
- Finally, a range of tools are required to help managers get the best out of their resources when developing large requirements models.

4.5.4 Analysis Style: High Level Decisions

OO ACT ONE is still in its infancy. However, already the research has identifed different ways of applying the *skeleton* method, which collectively can be considered to define different object oriented analysis styles. These styles are characterised by a sequence of high level decisions which are taken when the *skeleton* method is applied.

The analyst's job is to achieve a mutual understanding of problem domain structure with the customer and to record it in a meaningful way. Analysts can influence the representation of problem domain structure through interaction with the customer.

4.5.4.1 Achieving a Mutual Understanding of the Problem Domain

The structure of the problem domain should never be compromised to make the recording of the requirements suit the analyst. An analyst, on identification of a complex compositional problem domain, may suggest a better way for customers to structure their understanding. This better representation may reflect a simplification, but if the customer does not agree with the suggested (de)composition then no changes should be made. Problems arise only when customers view a problem in a very convoluted way and therefore make it difficult to express their requirements in a comprehensible fashion. In such a case the analysts must 'educate' the customer and attempt to relay a better understanding of the problem domain (if there is one). When the customer and analyst cannot find a mutually agreeable way of understanding the problem then there is no simple way to express the requirements. Requirements capture is not complete until both customer and analyst are sure that they have a common understanding of the specification produced.

Analysts are free to chose any means of capturing customer requirements. Within the *skeleton* approach there is enough analyst freedom to give rise to specification styles. These styles are predominantly related to achieving a balance between:

- Inheritance and delegation.
- Re-use and coding from scratch.
- Extending class hierarchies and redefining existing classes.
- Deep-and-narrow and shallow-and-wide structures.
- Bottom-up and top-down development.

4.5.4.2 Inheritance and Delegation

Although subclassing and composition are very different concepts, it is common within object oriented analysis to confuse the two notions. It is the analyst's job to remove this confusion. In the process of recording requirements an analyst is often faced with the choice of specifying new behaviour by inheriting from an already existing class, or by delegating tasks to an already existing class, as a component of the new class. Clearly, if the customer can distinguish between these two mechanisms of re-use, and can identify which mechanism is appropriate in each case, then there is no problem. However, when a customer is unsure it is the analyst who choses the solution.

4.5.4.3 Re-Use and New Class Production

When behaviour is required that is 'similar to' or 'related to' an already existing class, the temptation is to work around the re-usable component. This can lead to additional problems if the work needed to include the predefined component is more than the work which would have been needed to generate the required behaviour from scratch. We recommend that existing classes be re-used only in two cases:

- the component to be re-used is identified as being a (sub/super)class of another which is already part of the specification
- the class to be re-used has been identifed as a structure component of a class already in the specification

This is a conservative form of re-use since we do not advocate adapting components to fulfil ones needs. A less conservative (adaptive) style is one in which nearly all OO ACT ONE code is produced from already existing components. In other words, pick a class as close to what is required as is possible and adapt it until it suits. Analysts can develop a style which reflects their attitude to re-use.

4.5.4.4 Class Changes and New Subclasses

A high level decision is often required when deciding whether to define a new class as a subclass of an already existing class or to directly extend the existing class. Each choice has its advantages and disadvantages:

- i) Using the subclass relationship defines an explicit relationship which is guaranteed by the subclassing mechanism; but it extends the class hierarchy which, in general, should be as simple and uncluttered as possible.

- ii) Changing the already existing class keeps the class hierarchy simple but there are no automatic guarantees that the class maintains its old behaviour. Thus, the change may affect other clients of the class (those classes which inherit behaviour from it or use it compositionally).

Adopting the first choice has many consequences for re-use and implementation. Large complex hierarchies make re-use much more difficult since it makes it harder to find the behaviour which one requires. Also, implementing such hierarchies often has major overheads [101].

4.5.4.5 Structures: Deep and Narrow or Shallow and Wide?

Structure appears in the OO ACT ONE composition and classification hierarchies.

I) Composition

OO ACT ONE `STRUCTURES` define the composition properties of groups of objects whose components are members of the same class. There are two types of style for simplifying the recording and presentation of compositional information:

- **Flat and Wide Compositions:**
  The number of levels of (de)composition is kept to a minimum.
- **Deep and Narrow Compositions:**
  The number of components in each `STRUCTURE` is kept to a minimum.

As with all analysis problems, the exact style of specification depends on the customer and their problem domain. In all cases, independent of style, OO ACT ONE compositions should never be too wide or too deep.

II) Classification Structure

Structure in the class hierarchy is amenable to the same kind of reasoning as the compositional structure. To simplify understanding of a class hierarchy it is recommended that a class should not have too many direct subclasses (children) or direct superclasses (parents). Further, a class should not be too many subclass relationships away from its root superclass.

In an OO ACT ONE specification a large number of separate class trees commonly make up a system class hierarchy. There is no benefit, in our experience, of restricting the number of trees in each case. In fact, there are many advantages in having many smaller disconnected trees rather than connecting together classes into larger trees of class relationships.

As with the compositional structure, it is up to the analyst to interact with the customer to achieve the best mutual understanding of the system being analysed. The analyst's own particular style of seeing the problem is sure to influence the way in which the customer views the classification requirements.

4.5.4.6 Bottom-up vs Top-down

Object oriented analysis is both bottom-up and top-down. Already existing classes are synthesised into the requirements model, whilst parts of the problem domain which are not well understood are deconstructed in an attempt to improve understanding.

An object oriented analysis technique must be flexible so that problem domains which are well understood can be analysed compositionally whilst those less well understood can be analysed decompositionally. A certain style of specification will result when an analyst is involved in the same sort of problems over and over again. Certain components will be used repeatedly and the way they are incorporated in the new specification will be standardised by a particular analyst. Contrastingly, analysts who face a wide range of problems will build up a style of specification built around method rather than components. These analysts will, with experience, acquire general techniques.

4.5.5 General Analysis Principles

The following is a list of principles which analysts should adhere to:

- The customer is central to analysis. The requirements model must be customer led.
- Keep things simple.
- Specify for re-use, and re-use pre-defined components wherever practical.
- Test any changes to the specification which alter customer and/or analyst understanding.
- Comment all non-functional requirements:
  - Unspecified behaviour and the reason for its appearance.
  - Nondeterministic behaviour.
  - Persistency within a pure OO ACT ONE specification.
- Record all analysis decisions which re-structure customer understanding, together with customer reaction to such changes.
- Never rush analysis: it is the most important part of software development.

4.6 FOOA and Object Oriented Design

4.6.1 Importance of Structure

Explicit structure does not make a specification into a design. The structures in an operational specification are independent of specific resources in an implementation environment, whereas designs actually refer to specific resource allocation in a final implementation. Furthermore, the identification of the structures, and the interaction and relation between them, is problem domain dependent: they are chosen for the way in which they model customer understanding.

This thesis argues that it is impossible within an object oriented framework to acquire problem domain understanding of requirements free of structural bias. Even if this was possible, it would

be very difficult to specify them formally since all formalisms introduce internal structure to decompose complexity[3]. Structure is the only solution to capturing the requirements of complex systems. The structure explicit in an OO ACT ONE specification is not a necessary part of the design or implementation, but it does act as a good basis upon which a solution can be developed.

4.6.2 Executable Models

Executable formal requirements models are easier to communicate with the customer. Executable specifications make rapid prototyping straightforward and automatic. They improve the process of customer validation by allowing walk throughs of dynamic system behaviour. Test traces have their limitations but program proving, the other main approach to validation, has other well known inadequacies too [3, 8]. Executable specifications are necessary, at least until the state-of-the-art in proving specification correctness is further developed. From a project manager's point of view, executable models provide additional advantages with respect to early results and accountability. An executable specification is a clear statement of the progress being made during analysis.

4.6.3 Constructive vs Unconstructive Specifications

The debate for and against constructive requirements models has been proceeding in a wide range of publications without any agreement between the two factions. The non-executable advocates concede the difficulties of relating specifications to a customer, but argue that better solutions should be sought other than presenting the customer with an executable model (see [64], for example). It is argued that, in general, a specification written in a notation that is not directly executable contains less implementation detail than an executable model. Matching such a specification to user requirements is, it is claimed, more straightforward since there are no additonal algorithmic details necessary for executability[4]. It is also claimed that executable specifications unnaturally hinder designers by constraining the possible choice of implementations [78]. This thesis argues that it is useful to provide a concrete structure in which designers can begin their work.

This thesis takes an executable approach to requirements capture. It is the only approach, currently, of making a formal object oriented analysis model which is amenable to customer validation. Executing a requirements model is a vital part of customer validation [131, 107].

4.6.4 Design and Design Transformations: A Preview

The structure of an OO ACT ONE specification is problem oriented, not necessarily implementation based. During design, the specification is subjected to transformations which preserve external behaviour characteristics but alter or extend the internal structure to yield an implementation oriented

[3] Axiomatic methods do not appear to bias structure in as extreme a manner as other specification methods, but these have proved to be limiting in the type of behavioural characteristics they can define.

[4] In the object oriented paradigm, this argument seems less convincing than when applied to traditional approaches to software development.

architecture for the system being modelled. Much work has been carried out in developing and automating these *correctness preserving transformations* (CPTs). The work by Partsch [96] argues the case for CPTs in some detail. In chapter 5, we define a set of CPTs which work on full LOTOS specifications (with the ACT ONE as it is generated from the OO ACT ONE requirements model). We develop only a small set of CPTs, but these suffice to illustrate the general principles.

Using CPTs means that problem domain structure need not be compromised in a requirements model. The initial requirements can be optimized for clarity, re-use, maintainability and, above all, customer accessibility. Design transformations manipulate this structure to achieve implementation ideals based on efficiency, use of available resources and current programming practices. For an object oriented formal development approach to become widely used, CPTs must become standardised elements of design, and tool support must be provided to control the sequential application of such transformations.

Chapter 5

Formal Object Oriented Design (Using LOTOS)

This chapter is structured as follows:

- **Section 5.1: Introducing Design**
  This section introduces design. It argues that design is difficult because it is a combination of artistic and scientific abilities. Design quality is introduced, and the important difference between functional and nonfunctional requirements is briefly explained. Finally, software design is introduced: a short historical background is given, together with the identification of problems unique to software design and statement of intent to reuse work in other design areas, whenever possible.
- **Section 5.2: Learning From other Design Areas**
  This section compares software design and other types of design. Software design is shown to have the problem of coping with change. Design principles and techniques, common to all design areas, are identified: the importance of language, the role of structure, the advantage of re-use and the necessity of testing. Finally, this section identifies engineering as the area outside computing most closely related to software design.
- **Section 5.3: Object Oriented Software Design**
  Section 5.3 examines object oriented software design. Initially it gives an overview of software design by listing a set of general software design criteria and principles. Then, an explicit definition of the roles of object oriented designers is proposed. Finally, testing, verification and correctness preserving transformations (CPTs) are introduced.
- **Section 5.4: Object Oriented Design with LOTOS**
  This section considers object oriented design with LOTOS. We argue that LOTOS is a good candidate as a formal design language, even though it was not developed for this purpose. The importance of balancing the roles of the process algebra and ADT parts of LOTOS is stressed. Finally, the problems in defining an object oriented style of specification in LOTOS are considered.

- **Section 5.5: FOOA As Input To Formal Object Oriented Design**
  Section 5.5 formulates the initial step from object oriented analysis to object oriented design. Four different translations for mapping from OO ACT ONE to full LOTOS are considered. An object oriented interpretation of the first LOTOS specifications, produced using two of these transformations, is then given. The initial step from analysis to design is shown to be concerned with making concrete the semantics of object communication and interaction. Two models of control flow are chosen for particular attention: a remote procedure call model and a parallel model.
- **Section 5.6: Correctness Preserving Transformations (CPTs): Formalising Design**
  After introducing CPTs with respect to design in general, and design with LOTOS in particular, the fundamental concepts are reviewed: design trajectory, implementation relation, verification and CPT formulation. Internal and external properties are distinguished and this leads to a simple separation of CPTs into functional and nonfunctional categories. The importance of well defined non-standard semantic views of LOTOS (and their graphical representation) is re-iterated. Finally, the CPT design trajectory is introduced as forming the basis of an ideal object oriented design method, which this thesis goes a small step towards achieving.
- **Section 5.7: A Set of Object Oriented Design Decisions as CPTs**
  This section formulates five types of transformation for application during the design stage of FOOD: static structure expansion (decomposition), compositional restructuring for re-use, restructuring for distributing control, removing explicit nondeterminism and removing parallelism. In each case, the correctness of the transformations is discussed.

5.1 Introducing Design

Design, in general, is viewed as an artistic or creative process which combines natural ability with experience. It is found in many spheres of human activity, but it is far from being well understood and often seems inaccessible to the layman. The principles and practices which are applied to the design of software have a strong affinity with more traditional engineering: there is a subtle blend of scientific criteria with intuitive decision making.

The question of why design is difficult needs to be addressed. Most complex systems seem to have been built for a particular purpose. The designers of such systems obviously have this purpose in mind throughout the whole design process. In a sense, this type of design can be said to be *targetted*.

5.1.1 Design: The Creative Process

Design is a creative process concerned with decision making[1]. Designers look for solutions to problems. They search a solution space to arrive at a final design. The way in which the search is carried out may be methodical, but never deterministic. To design is to blend the old with the new: designers

[1] Thus, to do something by design means to do it by choice.

must use their experience and previous work (the old) to find a solution to their problem (the new). The creative side of design can be categorised as mixing three different modes of work:

- 1) Creating new components which are variations on already existing components and combining these new components in well-accepted ways (or structures).
- 2) Finding new ways of using components or combining components.
- 3) Gaining insight into a problem and building a design (component) to utilise this insight.

Most designers work in the first mode. For example, car designers create new cars by designing some new parts, utilising existing parts, and combining them in well established ways. Fewer designers work in the second mode. In the construction industry, for example, buildings with original structure (or layout) can be created from standard components. Building designers are aware of the way in which standard components can be combined and tend to concentrate on structure rather than individual components. The third mode of design is the rarest — perhaps these types of designers are better termed inventors?

5.1.2 Purposeful Design

In purposeful design, the designers have some goal to aim for and this goal is evident throughout the design process. Designers are involved in each design step in an attempt to reach thir goal. Central to purposeful design is the customer requirements. There are two extremes to the way in which designers can develop understanding of the requirements:

- Designers perform their own problem analysis to develop an initial requirements model.
- Designers accept a requirements specification in which the requirements are completely and consistently recorded.

In practice, design occurs somewhere between these two extremes. This thesis argues that design should not involve an analysis of the problem domain, although it does involve analysis of the requirements model. It is the role of designers to restructure the requirements model to best use the resources in the target implementation environment.

5.1.3 Design Quality and Criteria

A design provides, as an end product, one possible solution to a problem. The design describes the structure of the solution by defining:

- A set of components.
- The relationship between components.
- The method of construction, i.e. a means of realising the component relationships.

Given a design, it is necessary to be able to assess its *quality*. In other words, what is required is a set of criteria by which a design can be judged. The first and foremost test must be whether it fulfils

the requirements as specified by the analysts. Designs which do not fulfil their requirements are said to be *unacceptable*.

Requirements are traditionally divided into two groups: functional and non-functional (for a more complete analysis of the difference between these groups see [67]). Functional requirements are those specified in the requirements model. Non-functional requirements are usually concerned with costs, physical constraints, previous practices, political issues, etc .... A design which fulfils both type of requirements is said to be *acceptable*. Given two, or more, *acceptable* designs we must ask how a designer chooses between them. In such cases it is difficult to be objective. To say that one *acceptable* design is *better* than another is a subjective statement based on the (usually informal) criteria upon which judgement is made. In some less technical environments it would be called *taste*.

5.1.4 Introducing Software Design

5.1.4.1 A (Very) Brief History Of Design

In the beginning, 'programmers' analysed, designed and coded (they still do, in some instances). The 1970's saw the beginning of structured programming [45, 127, 44], which identified the need for a method to software production (coding). Structured programming evolved into the widely used set of different, though fundamentally similar, structured design methods. By the 1980's, structured design methods were widespread and their usage well documented [80, 94, 27, 36, 51, 41]. At this point structured designers designed and analysed[2]. The appearance of object oriented analysis and design methods, in the late 1980's and early 1990's [31, 25, 26, 13, 101, 84], were a consequence of the acceptance of object oriented principles within the programming community, and the transfer of these principles to the earlier stages of software development.

5.1.4.2 Software Design: Too Difficult For Words?

Software designers are faced with a unique set of problems:

- The requirements they are given make up a set of the most complex systems ever created by man.
- Software requirements are dynamic.
- Software design tools and methods are accessible to anyone with little experience in software development. When *bad* designers apply such methods, it is often the methods which get blamed for the resulting chaos. Software design is a complex process. The tools and methods are important, but designers must understand the underlying complexity of what they are being asked to produce and the principles behind the methods they employ. This is not always the case. Complex software designs can be produced very simply, but complex software designs which fulfil their requirements are not so simple to develop.

[2]Many of them also coded.

- Software designers work in a highly dynamic environment. As hardware capabilities improve and software requirements grow, the tools which designers are expected to use become increasingly more powerful (and complex). Designers are faced with a dilemma. They can either:
  - Stick with one method, become familiar with it and work within its limitations.
  - Continually evolve their methods to cope with all the new research on software development.
  - Change methods and techniques to suit the problem at hand.

  Each of these approaches has its advantages and disadvantages. Clearly, no matter which choice is made, designers cannot be oblivious of the dynamic nature of their working environment.

5.1.4.3 Software Design: Help Is At Hand

Software designers have two advantages over other types of designers. Firstly, software is pliable and can be manipulated so much more easily than other more concrete designs. Secondly, many of the other problems facing software designers have been faced by other types of designers in a wide range of problem domains. Software design is a new discipline, but many of the same principles and techniques used in other design areas are applicable in software development.

Design, in general, is about understanding requirements, understanding solution space, transforming structure and verifying design against requirements. A design method helps to co-ordinate these activities.

5.2 Learning From Different Design Areas

5.2.1 Allowing For Change: A Unique Problem

Software design is the only discipline in which designers expect the requirements they are given to continually change. In other areas, designers may be asked to extend their designs to incorporate new requirements, but only in computing are designers regularly expected to change their designs to accommodate requirements alterations. In other areas, designers consider such changes a major problem. In software design, such changes are the norm.

Perhaps wrongly, software engineering is seen as being inherently flexible: it is all too easy to change a few lines of code! This is true, but it is not easy to control the changes and to understand their consequences. The extremely pliable nature of software is both an advantage and a disadvantage. The dynamic nature of software requirements is a persistent problem. Changes in requirements occur during the design process, through the coding, and after the "final" product has been completed. Correcting mistakes and extending the software, otherwise known as *maintenance*, is an unending process. It is the compliancy of software which makes this possible. Problems arise because this is adversely taken advantage of by programmers. Complex software systems are prone to being made incomprehensible by uncontrolled change. Designers can, and should, play an important role in preventing this from happening.

The object oriented development strategy advocated in this thesis enforces controlled change. Designers should not be given a system (or parts of a system) to design until the requirements are fully understood and unlikely to change. In this way designers can work independent of analysts. Certainly mistakes will be made, but these are the analysts' responsibility, not the designers. The correction of mistakes in the requirements model should be easy to filter through to the design, provided the initial transformation to design maintains a mapping between components, and the subsequent design decisions are well documented. Similarly, if the requirements model is defined with modifiability and extendibility in mind, the resulting designs should also exhibit these features (to some degree).

5.2.2 Identification of General Techniques and Principles

The central theme of design is structure management. Like analysts, designers manage complexity by enforcing structure on the way in which behaviour is represented. (The difference between these two development stages is that analysts work solely with problem domain structure, whilst designers work with structure which, at each stage of design, is a fine balance between problem domain and solution domain architectures). Consequently, some of the techniques and principles evident in analysis are also evident in design:

- The importance of notation (language of expression).
- The importance of structure: hierarchical and configurational.
- The role of re-use.
- The ability to test a model against requirements.

5.2.2.1 Design Language

Design is concerned with communication. Consequently, the design language is fundamental to the design process. Language is any means of communication through the use of conventional symbols. Everyone is familiar with their own natural language. What is surprising is the number of other languages from which people can acquire some information: for example, maps of all various types, architectural plans, furniture construction instructions, mathematical equations, chemical formulae, pages of music, chess notation, recipes, etc ....

Certainly there will always be a relationship between natural language and other forms of representation since natural language shapes the way in which we can think. There is always a good reason why natural language is not used to communicate certain types of information:

- Natural language is not good for communicating spatial properties.
- Natural language is too expressive and often what is required is a simpler notation.
- Natural language is open to interpretation.

Languages are developed to make the recording of certain information simple, elegant and concise, whilst making the representation of other information very difficult. Language is the most important tool for abstracting away from unimportant information. Abstraction is fundamental to all areas of design.

5.2.2.2 Structure: (de)composition

The doctrine "divide and conquer" is central to all human activities and can be applied particularly well to design processes. Targetted design involves taking a set of requirements and producing an object which can be said, in some way, to fulfil these requirements. Designers decompose a requirements model to achieve a structured understanding. All designers follow (or advocate in part) repeated application of the "divide and conquer" maxim. Structured requirements aid the process of decomposition by providing a natural means of initially decomposing a problem. Design is also about composition: putting components together to form new structures applicable to the solution domain.

5.2.2.3 Re-use

All designers re-use elements of their working environment: experience, methods, structures and components. Software designers must learn from other areas in which re-use is prominent. This learning can be passed on to a community of designers when it is incorporated in a general design method. The main form of re-use advocated for the object oriented design stage of FOOD is the repeated application of correctness preserving transformations. Designers are then re-using well defined ways of taking designs from the abstract to the concrete. Component re-use is more prominent in the analysis and implementation stage of FOOD.

5.2.2.4 Testability

All designs must be tested against requirements. In many design areas these tests are informal and difficult to guarantee. Software designs are very difficult to test because the requirements which they are developed to fulfil are usually very complex. Rapid prototyping and modelling are well accepted ways of testing. This thesis shows that formal object oriented design is particularly well suited to rapid-prototyping.

5.2.3 Software Design and Engineering

Software design is often called software engineering. This is a reflection of the similarities between the roles of software designers and engineers of all disciplines. Engineering, in general, has well established methods which are governed by physical laws. Engineers learn to employ standard means of representation. Systems being engineered can, in general, have their approximate behaviour determined through analysis of the design documentation. Standard mechanisms and tools exist for constructing solutions from many types of engineered design.

Rather than expanding on the engineering analogy, this thesis acknowledges that software development should be extended to reflect the practices evident in engineering disciplines. In particular, formal techniques of software development are lacking in development method. *Formal methods* are really a set of models and tools which are usually distinct from a particular method (way of using the models and tools). Engineering balances method with the underlying models (based on physical laws and mathematical systems) in a way which software designers should attempt to emulate.

5.3 Object Oriented Software Design

5.3.1 Overview of Software Design

Rather than reviewing a wide range of particular software design methods, whose main role is to go from problem domain structure to solution domain structure, this section borrows an approach by Meyer [84] by identifying criteria for evaluating design methods and stating the principles upon which good software design should be based.

5.3.3.1 Design Criteria

The criteria for judging design methods are similar to the criteria for judging analysis methods (see chapter 2):

- Design languages must incorporate explicit structuring mechanisms.
- Design methods must encourage a structuring in which there are components to match elements in both the problem domain semantics and solution domain semantics.
- Design methods and models must combine in a consistent and coherent fashion.
- Design methods must be flexible to allow for designer creativity.
- Design methods must facilitate re-use and encourage the rapid development of experience.

5.3.3.2 Design Principles

When communicating complex ideas, it is important to keep things as simple as possible. The underlying principle is therefore to make the final design only as complex as the requirements demand, and to make the process of achieving this design as simple as possible to understand.

Most software development methods recommend the following as a means of reducing complexity:

- Strong cohesion.
- Weak coupling.
- Well defined interfaces between components.
- Encapsulation of components.
- Limiting the number of components at each level of abstraction.

Re-use is also a prominent feature of design. It is argued that re-use aids understanding. The different types of re-use in software engineering are well documented: [59] provides a good overview of the subject. This thesis advocates re-use at all stages of software development.

5.3.2 Comparing Object Oriented Design and Object Oriented Analysis.

We have argued that object oriented development is superior to other development methods because of the conceptual integrity between problem domain and solution domain. In particular, we have stated that the problem domain structure should be present in the design. This begs the question:

what does an object oriented designer do if the structure is maintained throughout the whole object oriented development method? In answer to this question, we identify four responsibilities of an object oriented designer:

- The removal of nondeterminism.
- The concrete realisation of the abstract object oriented concepts as specified in the requirements model.
- The restructuring of requirements to suit an implementation environment.
- Verification of design against requirements.

Each of these responsibilities are examined in the following sections: 5.3.3 to 5.3.5.

5.3.3 Removing Nondeterminism

In chapter 5, two types of nondeterminism arise from the specification of probabilistic behaviour and implementation freedom in the requirements model. Designers must remove both types.

Removing probabilistic nondeterminism involves specifying the probabilistic requirements in some standard way which is amenable to immediate coding. This type of design step is not examined in any detail in this thesis. The analysis method in chapter 4 identifies a means of recording the probabilisitic behaviour which separates probabilistic properties from other behavioural concerns. Consequently, these less-abstract properties can be abstracted away from during design. It is beyind the scope of this thesis to investigate the design and implementation of probabilistic behaviour.

Removing the nondeterminism due to implementation freedom in the requirements is one of the major responsibilities of design. Analysts are encouraged to offer sets of alternate permissable behaviours from which a designer is required to choose one particular solution. One way of removing this type of nondeterminism is specified by the *Rend* CPT, defined in section 5.7.4.

5.3.4 Realising the Abstract Object Oriented Model

The object oriented requirements model must not specify implementation concerns. An analyst is not concerned with whether the objects are going to be implemented as, for example, concurrent processes (on distinct processors), imperative records or Eiffel class instances. At the simplest level, the analysis model does not even state how objects communicate with each other, or how their state is realised. The designer must know the way in which the abstract semantics in the requirements model can be mapped on to the target implementation language semantics.

5.3.5 Restructuring The Requirements To Match An Implementation Environment

Restructuring has two main goals:

- To facilitate re-use of implementation code.
- To take advantage of the high level language constructs in the implementation language.

5.3.5.1 Restructuring For Re-use

A flavour of the designers role in the process of restructuring for re-use is illustrated by the two examples in figures 5.1 and 5.2.

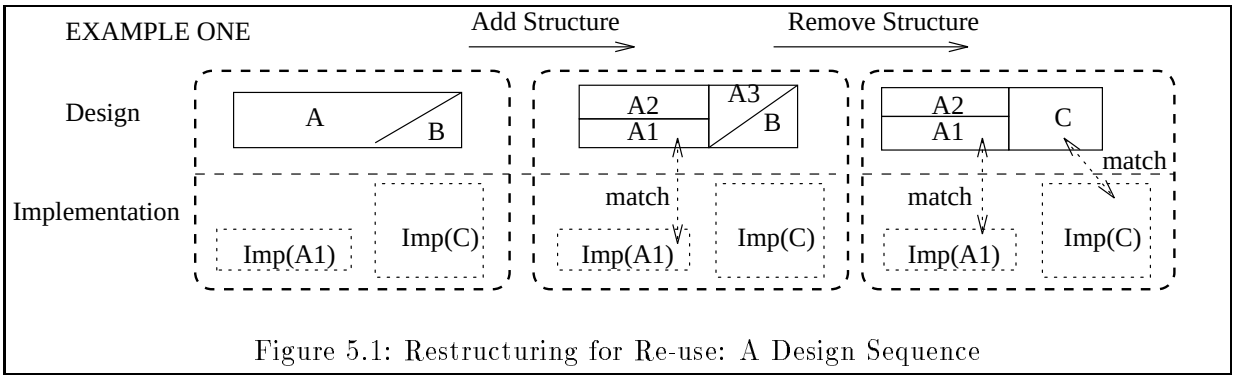


Figure 5.1: Restructuring for Re-use: A Design Sequence

In example one, in figure 5.1, the designer has identifed two re-usable implementation components which provide part of the behaviour in the first design. Implementation component `A1` provides part of the behaviour of design component `A`. Implementation component `C` provides all of the behaviour of design component `B`, together with some of the behaviour of `A`. The designer can restructure the design to utilise the already existing components, and make the job easier for the coders. The first design step is to decompose `A` (i.e. add structure) so that one of the design subcomponents matches the implementation component `A1`. In this step the designer also structures the other subcomponent into parts (`A2` and A3) in anticipation of the next composition step. The next design step is to compose design components `A3` and `B` to make a match with implementation component `C`. At this point of the design, the implementers must code only one new component, an implementation of `A2`, and combine three components, namely `A1, A2` and `C`.

Example two, in figure 5.2, represents a branching in the design.

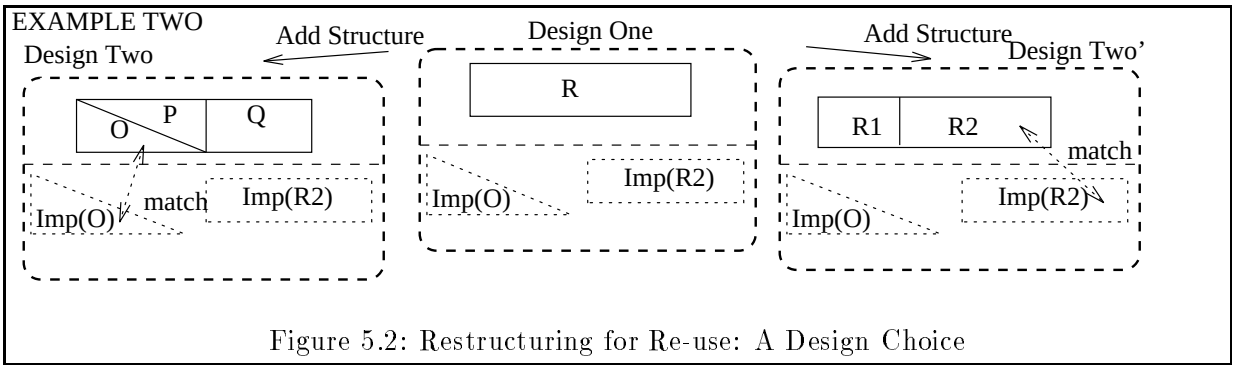


Figure 5.2: Restructuring for Re-use: A Design Choice

The designers are aware of two components which each provide a part of the behaviour specified in the design. However, the two components cannot be used together (perhaps because of some constraints in the implementation environment). The designers would like to re-use at least one of them, and so must choose which design trajectory to follow (both involve adding structure).

Example two also illustrates a more complex design trajectory. Consider `Design Two` as the initial

design. A 'small' part of the design, namely `O`, is directly implementable (i.e. a piece of pre-written code exists). However, a different implementation component, represented as `Imp(R2)`, provides a large amount of behaviour of the design components `O, P` and `Q`. The designer ignores the obvious re-use of component `O` and restructures the design to finish with stage `Two'` (via stage `One`). These types of restructuring design steps are facilitated by the correctness preserving transformations defined in section 5.7.1. and 5.7.2.

5.3.5.2 Restructuring For Implementation Language Utilisation

The implementation language which the designer is aiming towards may offer high level constructs which the designer must attempt to utilise. The implementation environment may offer, for example, concurrency, distributed processing, multiway synchronisation or sharing. Designers must structure their designs to make these properties explicit in the design, so that there is a better match with components in the implementation language. The correctness preserving transformation *Dist*, specified in section 5.7.3, performs this type of role.

5.3.6 Verification and Correctness Preserving Transformations

5.3.6.1 The Need For Formality

The most important role of the designer is to verify that the designs produced fulfil the requirements. Formal requirements models and formal designs are essential in improving the verification process, and making the customer (and software producer) confident in the final product. Given a formal requirements model, the general verification of any given formal design against the requirements is very difficult and, except in the most simple circumstances, impossible to guarantee in the present development environment. However, all is not in vain. Designers can be encouraged to follow a design trajectory in which the design evolves in a number of stages. Each stage can then be verified against the original requirements model.

The design trajectory method works for the following reason. The first design can be verified against the requirements. Then this design can be manipulated to achieve the next design, and so forth. Provided each design is verified against the previous one, by induction, the final design is verified against the original requirements. (Note that the correctness relationship must be transitive.)

Following a design trajectory does not make the verification process at each stage of the design any easier. For example, a designer can make any number of complex changes between each stage. In such cases, designers are back to the original problem: verify any given design against the requirements model. Consequently, most design trajectories restrict the type of changes that can be made at each design stage.

Correctness preserving transformations (CPTs) take this restriction one step further. Designers which restrict themselves to using CPTs do not need to do any verification at all! The CPTs are defined, and proven, to guarantee that, if a design, $D1$ say, is transformed by a CPT into another design, $D2$ say, then some properties of $D1$ are guaranteed to be fulfilled by $D2$.

5.3.6.2 The Advantages of an Object Oriented Environment

The specification of CPTs is not easy. Designs go from the abstract to the concrete. CPT development, in an object oriented framework, is aided by consistency in the underlying semantic model at all levels of abstraction. A major difficulty is in identifying transformations which are used throughout the design process, and formally defining them as CPTs. Another difficulty in defining such a design trajectory is in verifying the first design against the requirements model, especially if the semantics of the two models are very different.

The approach taken in this thesis means that the original requirements model is directly incorporated in the first design. The ACT ONE model of the object oriented requirements is carried over to the initial full LOTOS design specification. Further, the CPTs are defined only on the object oriented LOTOS specifications which, by definition, contain the original ADT requirements model. This makes the specification of CPTs much easier than in the general case, where they must be defined on the domain of all valid LOTOS specifications.

5.4 Object Oriented Design with LOTOS

5.4.1 Design in LOTOS

5.4.1.1 An Overview Of Design and Verification

LOTOS may not appear, at first glance, entirely appropriate for formal software design. It was developed for use in the area of standards (particularly service and protocol specification) and consequently its semantics is abstract: the specification of standards must be implementation independent so that manufacturers are not restricted in the way they can develop products to fulfil the standards. Manufacturers must permit testing of their products against standards, without having to give access to internal details. Thus, most conformance relations in LOTOS have been defined to be observational [10]. Verification of LOTOS designs is not restricted to 'black box' testing. Designers have complete views of the designs before and after each design stage. Thus, the conformance realtionships used by designers need not be restricted to being observational in nature.

5.4.1.2 Design Is About Structure

Inherent in the design process is the notion of structure. Design is the process of creating a framework upon which a set of requirements can be realised. Design languages must have explicit structuring mechanisms. An advantage of software design is that the requirements are (formally) specified and as such will be structured. This gives designers an initial framework from which they can gain understanding of the requirements, and on which they can start to create a design. An object oriented requirements model is even more advantageous since the requirements model structure is likely to provide a good framework on which to start the production of an object oriented implementation.

There are two types of structure which are fundamental to design:

- **Hierarchical**
  Hierarchical structure is one in which elements of the design are related by an ordering. In object oriented design, subclassing between classes and composition between objects define two different types of hierarchical relationship.
- **Configurational**
  Configurational structure is one in which elements of the design are connected by relationships which do not have an implicit ordering.

LOTOS provides facility for defining both types of structure property. Consequently, LOTOS fulfils one of the necessary (but not necessarily sufficient) conditions for a language to be suitable for software design: it must support structured specifications.

5.4.1.3 Software Design Is About Controlling Change

Another aspect of software design is that the designs must be manipulatable. Designs must be amenable to change so that a design trajectory is simple to follow. This is certainly true of LOTOS, but pliability is only half the story. The manipulations must be controllable: the reasons for making a design change must be fully understood, and the consequences deterministic.

All structured design methods (including object oriented approaches) argue that their techniques are advantageous because changes can be kept as local as possible. Often they identify the need only to change one component of the design at a time. Certainly LOTOS offers this type of local change facility: ACT ONE sorts and LOTOS processes can be treated as modular elements. However, design is not just about localising change. For example, a high level design decision might be to change the server-client communication model throughout the whole design (in response to a change in the target implementation language, for example). Such a change is inherently global. The use of CPTs helps to make such changes in a controlled way. LOTOS specifications are formal and are therefore amenable to controlled global change.

A final requirement of a design language, with respect to structure and structure manipulation, is that it can express behaviour at very different levels of abstraction. The initial design must be very close to the requirements model, whilst the final design must be very close to an implementation model. A language which can represent a range of behaviours, from abstract to concrete, is called a wide-spectrum language. Software design languages which are used throughout a comprehensive design trajectory must be wide-spectrum. LOTOS is such a language.

5.4.1.4 Designs Must Be Verifiable

Designs must be verifiable against requirements. LOTOS, as a formal language, is open to mathematical verification against a formal requirements model. However, formality alone is not sufficient. Formal methods are dependent on tool support. It is impossible to verify even the simplest set of requirements by hand. Consequently, we require that a formal design language must have a reasonable tool support, with further support in the forseeable future. LOTOS fulfils this requirement.

5.4.2 Abstract Data Typing in LOTOS

LOTOS contains an abstract data typing language based on ACT ONE. Chapters 2 to 4 illustrate the power of ADTs to express structural properties (albeit in an object oriented framework of interpretation). Chapter 4 also identifies the weakness of the ADT approach to modelling: ADTs are not good for for recording communication, synchronisation, timing, concurrency and distribution properties. Quite deliberately, in this thesis, these high-level design features are abstracted away from during requirements capture. The requirements model says *what* rather than *how*. ADTs do play a major role in LOTOS designs: they maintain the underlying abstract behaviour whilst the process algebra is used to define the more concrete high-level design properties.

5.4.3 The Process Algebra in LOTOS

LOTOS is also constructed from a process algebra. These languages can, by themselves, record requirements in a highly structured fashion, which can then be interpreted as high-level designs. For example, processes can be decomposed into component processes which combine (using the parallel operators). This type of decomposition gives rise to hierarchical and configurational relationships structure. The components' interaction during event synchronisation is configurational, whilst the (de)composition is hierarchical. There has also been much work in defining hierarchical relationships between processes, based on the behaviour they offer. For example, there are many inter-process relationships which attempt to model subclassing and implementation properties, see for example [9, 33, 8].

Given the structural expressiveness of the process algebra part of LOTOS alone, we must question why the ADT part is required. The ACT ONE is necessary for defining (or modelling) the following:

- Parameterised behaviours in the shape of parameterised process definitions.
- Systems with explicit state components, also in the form of parameterised processes.
- Non-constructional properties in the form of preconditoned (guarded) behaviour.
- Structured events, which are necessary to model value matching, value passing and value generation.
- Process functionality.

LOTOS without the abstract data typing, often called basic LOTOS[3], can specify only a limited range of behaviours. This thesis uses the ACT ONE specification, as it is generated from the object oriented requirements, in conjunction with the process algebra.

5.4.4 Balancing Processes and Types in Design

One of the main problems with designing in LOTOS is in achieving the correct balance between ADT specification and process algebra specification. This thesis advocates using the ADT part as

[3]The terms 'basic LOTOS' and 'full LOTOS' are used to distinguish between LOTOS specifications with and without ADT parts, respectively.

a functional behaviour carrier and the process algebra as a high level structuring mechanism for specifying communication, synchronisation, concurrency, etc.... Generally, in LOTOS specifications, the balance between the two parts of the language is not so clearly defined.

A good introduction to LOTOS specification and the different roles of the process algebra and ADT parts is given in [120]. It also introduces the notion of specification style, and the way in which different specification styles place different emphasis on the roles of each part of the language. In general, different styles are best suited to specification at different levels of abstraction. Consequently, one approach to design is to specify a number of transformations between LOTOS styles. However, it is not yet possible to automate the whole design trajectory in the form of a complete set of CPTs. Rather, designers will be expected to directly interact with, and perform manipulations on, the LOTOS designs. Designers should not be asked to cope with LOTOS specifications written in many different styles. Jumping between different conceptual frameworks does not aid the design process.

This thesis advocates a design trajectory in which the style of LOTOS specification remains consistent. The specifications progress from the abstract to the concrete, but the underlying object oriented conceptual framework is maintained. It is the balance between the amount of behaviour specified in ACT ONE , and the amount specified in the process algebra which changes as the design evolves. In this way there is a clear reasoning behind the balance at any particular point in the design process.

LOTOS specifiers often have their own preferences in the way in which they use the ADT and process algebra parts. This favouritism is probably a consequence of their familiarisation with, and understanding of, the two different types of semantics underlying the two languages. It is not good that specification designers can influence the structure of their designs in a way which is not amenable to analysis. Another reason why the balancing between ADT and process algebra parts is so subjective is that there are no well accepted methods for developing formal specifications in LOTOS. There are plenty of tools for automation, validation and verification, and a wide range of example specifications, but there is little advice (and tool support) for the actual process of constructing the specifications. In particular, there is a real lack of management support[4]. Consequently, there are no existing methods (or tool support) for combining the ADT and process parts in a consistent and coherent way. Both parts can be used to record structured information but, without a method (or guidelines, at the very least), it is difficult to say which types of behaviour should be defined using which part.

5.4.5 Defining an Object Oriented LOTOS Style of Specification

Many attempts have been made to define object oriented (or object based) styles in LOTOS, for example [118, 100, 81, 24, 35, 6]. There is a vague concensus of understanding concerning the mapping between LOTOS constructs and the object oriented paradigm:

- Processes define classes of behaviour, usually of type noexit.
- Objects are instances of processes. The state of an object is represented by the parameterisation of the process.

[4]Perhaps this is the real reason why formal methods, like LOTOS, have not become accepted in industry?

- Objects (process instances) service requests through interaction at their external gates.
- The passing of parameters (input and output) between client and server corresponds to event synchronisation (and value agreement) at these gates.
- Subclassing is some relationship between process instances (objects).

There are two problems with this informal correspondence:

- The ADT part does not seem to have an active role.
- There is no relationship between classes of behaviour: the parameterised process definitions. The behavioural relationships are defined between process instances, and thus there can be confusion in differentiating between object and class.

In the object oriented LOTOS designs developed in this thesis, the ADTs play a very important role: they maintain the behaviour specified in the requirements model.

In object oriented design, there needs to be a clear distinction between classes and objects. When using LOTOS, confusion arises because process instances define a set of behaviours and this gives the impression that such a set is a class. This thesis argues differently. In our object oriented interpretation of LOTOS, each process definition corresponds to a class. Process instances correspond to objects and the set of behaviours defined by each object represents the set of valid implementations. The ADTs define the underlying behaviour of each class and the process algebra part of each class definition defines the high-level properties of the system.

The fact that one standard object oriented style of LOTOS specification has not been formally defined and well-accepted is indicative of the problems in the object oriented community. This thesis argues that the problem is not with LOTOS, it is with the inherent informality in object oriented systems and the many different interpretations of the object oriented concepts.

Object oriented concepts are not well understood, although there has been some recent work in defining object oriented semantics [47, 95, 129, 37, 29, 130]. These semantics were not chosen for use in this thesis because:

- They do not take a natural *state-transition-system* view of objects and classes.
- They do not recognise the importance of an object offering a constant interface during its lifetime.
- They do not match our intuition of objects and classes at all stages of software development.

Defining an object oriented semantics in LOTOS appears, at first glance, to be a rather appealing solution to the problem of informality in object oriented designs. However, such a solution is not general enough since the object oriented model so produced is too concrete for use during analysis. For example, even something as simple as the client-server communication model cannot be specified in the process algebra part of LOTOS without straying into implementation details. Such a model inherently restricts object oriented implementers to: synchronous or assynchronous communication, concurrency or sequentiality, distributed or centralised control etc.... Full LOTOS is a good language to specify object oriented models at a concrete level, but using the process algebra during analysis and requirements capture may be too soon.

5.5 FOOA as Input to Formal Object Oriented Design

5.5.1 Generating Full LOTOS from the Requirements Model

The requirements model, as specified in OO ACT ONE, and realised in the translation to ACT ONE, is an abstract statement of the behaviour a system is required to offer. The object oriented requirements say *what* is required rather than *how* a solution should be implemented. This is illustrated very well when the initial mapping to full LOTOS is considered.

Given a set of requirements of a system specification in an OO ACT ONE class definition, there are a number of ways in which these requirements can be translated to an initial abstract LOTOS design specification. Common to all such translations must be the retention of the ACT ONE class definitions (as represented in ACT ONE) in the full LOTOS code. This makes verification of the initial design against the requirements model straightforward.

Object oriented designers must initially identify the communication aspects of the way in which the underlying object oriented behaviour is to be fulfilled. The designers of a system must decide how the behaviour is to be offered at its external interface (and what this external interface should look like). This simple decision can affect the rest of the design process. Identifying an object oriented communication model and specifying the translation from OO ACT ONE, is not simple. There are a number of alternative models and a number of ways in which these can be specified. Four of these alternatives are examined in the following sections (5.5.1.1 to 5.5.1.4). The list is not exhaustive and the ways of specifying the models are limitless.

5.5.1.1 Remote Procedure Call (RPC) Model

The RPC model is based on the principle that while an object is servicing a request, no more requests can be accepted. Consider such a specification for the well accepted `Stack` behaviour[5]. The `Stack` elements are arbitrarily chosen to be `Nats`. This same `Stack` behaviour is also used to illustrate the other initial design alternatives. The RPC `Stack` behaviour is defined in the `RPCStack` process, below.

RPC: Stack example one

```
process RPCStack[push,pop](SStack: Stack): noexit:=
(push? Nat1: Nat; RPCStack[...](.(push(SStack, Nat1))))
[]
(pop; pop! NatResult(pop(SStack)); RPCStack[...](.(pop(SStack))))
endproc (* RPCStack *)
```

This style of specification is useful when the target implementation language has a procedural communication/interaction semantics. The `RPCStack` clients must wait for the `Stack` object (`RPCStack` process instance) to finish servicing its current request before their requests are accepted. In effect,

[5] In this example, and all others that follow, non standard syntax for the specification of the process gate list is used. When a gate list in a process instance is to be specified exactly as the gate list in the process header then it is more concisely written as [...].

the RPCStack refuses to participate in service request events if it has not yet finished servicing its current request. Note that this *lock-out* does not occur when TRANSFORMER attributes are serviced.

5.5.1.2 Parallel Access Model (Ordered In)

In this communication model, an object can service all requests at any time in its life (i.e. there is no *lock-out*). The order in which the requests are serviced is the order in which they are requested. However, the order in which the replies are given back to the requesting environment may not be maintained. The LOTOS specification of the Stack behaviour in this communication framework is given in the PAMStack process definition, below.

Parallel Access Model (Ordered In): Stack Example 2

```
process PAMStack[push,pop](SStack: Stack): noexit:=
(push?Nat1:Nat; PAMStack[...](.(push(SStack, Nat1))))
[]
(pop; ((pop!NatResult(pop(SStack)); exit) ||| PAMStack[...](.(pop(SStack)))))
endproc (* PAMStack *)
```

The PAMStack process can always accept a push or pop request (i.e. participate in a push or pop event). The transformer attribute push, like in the RPC model, is served instantaneously: the resulting state transition (re-instantiation of the PAMStack process with new state parameter) is achieved without need for a sequence of internal events. The dual operator pop is defined in terms of two event synchronisations: the attribute request pop and the attribute response pop!NatResult(pop(SStack)). Unlike in the RPC model, the PAMStack process can accept other service requests between receiving a pop service request and returning the pop result. This is specified using the parallel operator (|||). The result of the pop request is offered in parallel with the behaviour of the PAMStack. A consequence of this communication model specification is that, since multiple results can be offered in parallel, results do not necessarily have to be popped off in the order in which they are requested. This type of property is, in general, undesirable. Consequently, we do not consider this model for use during FOOD.

5.5.1.3 Parallel (Ordered In Ordered Out) Model

In this model, an object can service all requests at any time. The order in which the requests are serviced is the order in which they arrive. The order in which replies are sent is also maintained by the serving object. The *Ordered In Ordered Out* Stack behaviour is specified by the ParStack process in example three.

This LOTOS specification is much more complex than the others. It is providing Stack behaviour wrapped between input and output queues. The StackIn and StackOut processes are parameterised on a Nat. These parameters are used to tag requests as they come in and guarantee the order of responses on the way out, respectively[6] The queueing of service requests and responses is achieved by the parallel operators in the specification. There is no explicit queueing behaviour defined in the

[6]The state parameters of the In and Out processes can be initialised to any value provided it is common to both.

Parallel (Ordered In Ordered Out): Stack example three

```
process ParStack[push,pop](SStack: Stack): noexit:= hide request, response in
StackIn[push, pop, request] (0) |[request]|
StackBody [request, response](SStack) |[response]|
StackOut [pop, response](0) where
process StackIn[push, pop, request] (ID: Nat): noexit :=
Reqs[push,pop,request](ID) |[request]| ReqController[request](ID) where
process Reqs[push,pop,request](IDsStackIn:Nat): noexit:=
(push? Nat1:Nat;
( Reqs[push, pop, request] (.(inc(IDsStackIn))) ||| (request!push!Nat1!IDsStackIn; exit)))
[]
(pop;
(Reqs[push,pop,request]( .(inc(IDsStackIn)))||| (request!pop!IDsStackIn; exit)))
endproc (*Reqs*)
process ReqController[request](ServeID:Nat):noexit:=
(request!push?Nat1:Nat!ServeID; ReqController[request](.(inc(ServeID))))
[]
(request!pop!ServeID; ReqController[request](.(inc(ServeID))))
endproc (* ReqController *) endproc (*StackIn*)
process StackBody[request, response](SStack: Stack): noexit:=
( request!push? Nat1: Nat?ID:Nat;
(StackBody[request, response](.(push(SStack, Nat1)))||| (response!push!ID; exit)))
[]
( request!pop?ID:Nat;
(StackBody[request,response](.(pop(SStack)))||| (response!pop!NatResult(pop(SStack))!ID; exit)))
endproc (*StackBody*)
process StackOut[pop, response](CountStackOut: Nat): noexit:=
(response!pop?NatStackOut:Nat!CountStackOut;
pop!NatStackOut; StackOut[pop, response](.(inc(CountStackOut))))
[]
(response!push!CountStackOut;
StackOut[pop, response](.(inc(CountStackOut))))
endproc (* StackOut *) endproc (* ParStack *)
```

ADT. The translation to the parallel (ordered in ordered out) LOTOS model (for conciseness we call this the Par model of communication) requires an ACT ONE sort `Nat`, to provide the unique identification for each request, and an ACT ONE sort with literal members `push` and `pop`, which by convention is named `StackServiceRequests`, in order to differentiate between internal service requests. In translation, all classes in the OO ACT ONE specification have their external attributes defined as literals in a *ClassName*`ServiceRequests` sort. All these sorts are defined in a global `ServiceRequests` type specification.

Further, the `Nat` parameter sort can be replaced by any sort which offers a means of allocating an infinite set of unique identifications. `Nat` was chosen for its simplicity.

5.5.1.4 Parallel Explicit Routing Model

In this model, an object can service all requests at all times. The ordering of servicing is maintained, but the ordering of replies is not. However, each service is tagged, by the environment, with a unique identifier. This means that the environment can control the ordering and guarantee that the replies get returned to the correct clients. The explicit routing model of `Stack` behaviour is defined by the `ExpStack` process, below.

Explicit Routing: Stack example four

```
process ExpStack[push,pop](SStack: Stack): noexit:=
(push? Nat1: Nat? ID: Nat; ExpStack[...](.(push(SStack, Nat1))))
[]
(pop?ID:Nat; (pop!NatResult(pop(SStack))!ID; exit) ||| ExpStack[...](.(pop(SStack))))
endproc (* ExpStack *)
```

This specification is similar to the `Stack` behaviour defined in example two. The only difference is that the requests are accompanied by an identification (`ID`) which must be provided by the client of the `Stack`. These `ID`s are then tagged to the `pop` replies. This type of communication model can be utilised in the design process to define internal communication. However, explicit routing is, in general, too concrete a model to be used in the initial design stages. Consequently, we do not consider it for use in FOOD.

5.5.2 Internal and External Communication

The four models, above, define external communication properties for classes of objects. Two of these, namely the RPC and Par models, are used to define two fundamentally different communication models. These models define only the external interaction between a client and a server. They do not specify the internal communication which occurs when an object is servicing a request. The reason for this is simple: there is no internal communication in the RPC and Par processes. These processes are not defined as interacting systems of component processes (i.e. they are unstructured[7]). In section 5.7 we examine a means of structuring LOTOS designs. The means of interaction between component processes of a system is said to be defined by the resulting internal communication model.

5.5.3 Defining the Mappings from OO ACT ONE to Full LOTOS

Given an OO ACT ONE class specification, `class` say, then we define two transformations for the generation of full LOTOS specifications of `class` behaviour:

- MakeRPC(`class`) produces the RPC LOTOS design of `class` behaviour in a `RPCclass` process definition.
- MakePar(`class`) produces the Par LOTOS design of `class` behaviour in a `ParClass` process definition.

[7] More precisely, all the composition structure is contained within the ADT part of the design.

Appendix E2 defines these two mappings. Central to each mapping is the inclusion of the ACT ONE requirements model of `class` behaviour.

In the remainder of this thesis, any process identifed as `RPCclassname` or `Parclassname` is assumed to have been derived from the OO ACT ONE specification of `classname` using the appropriate mapping.

5.5.4 An Object Oriented Interpretation of the Initial LOTOS Designs

5.5.4.1 Notation Conventions

The OOLOTOS[8] specifications follow the following syntactic conventions:

- Every class in the OO ACT ONE system requirements has an ACT ONE sort of the same name. These sorts are specified in the ACT ONE requirements model generated from the OO ACT ONE specification.
- An instantiation of the `system` class corresponds to an instantiation of an RPC or Par process.
- The gate list of the `system` class process corresponds to the list of transformer, accessor and dual attributes of the class. (The ordering in the OO ACT ONE specification is maintained in the LOTOS code.)
- The state of the `system` class process is identified by the variable `Ssystem`, a value of the ACT ONE sort `system`.

5.5.4.2 Processes and Objects

The relationship between OO ACT ONE classes, ACT ONE sorts, and LOTOS processes is illustrated in the top half of figure 5.3. The internal aspects of the process specifications expand out as design progresses: they are not represented in the figure. The structure of the problem domain is retained in the ADT part of the LOTOS designs. The CPTs can be used to transfer this structure to the process algebra for further manipulation. The bottom half of figure 5.3 shows the simple relationship between objects in the three different notations.

5.5.4.3 External Attributes and Servicing Requests

There is a direct correspondence between the gates of a process and the external attributes of the class which it is modelling. The hidden attributes of a class are not included as part of the external gate list of the class process and, consequently, the behaviour offered by the hidden attributes can be accessed only through the ADT specifications. The internal (nondeterministic) transitions in the requirements are modelled as internal events by the LOTOS **hide** operation.

The parameterisation of the external attributes is matched by the event synchronisations between the process and its environment. A service request is modelled, in the server process, as an 'input

[8] OOLOTOS specifications are defined to be those specifications which are derived from OO ACT ONE requirements models.

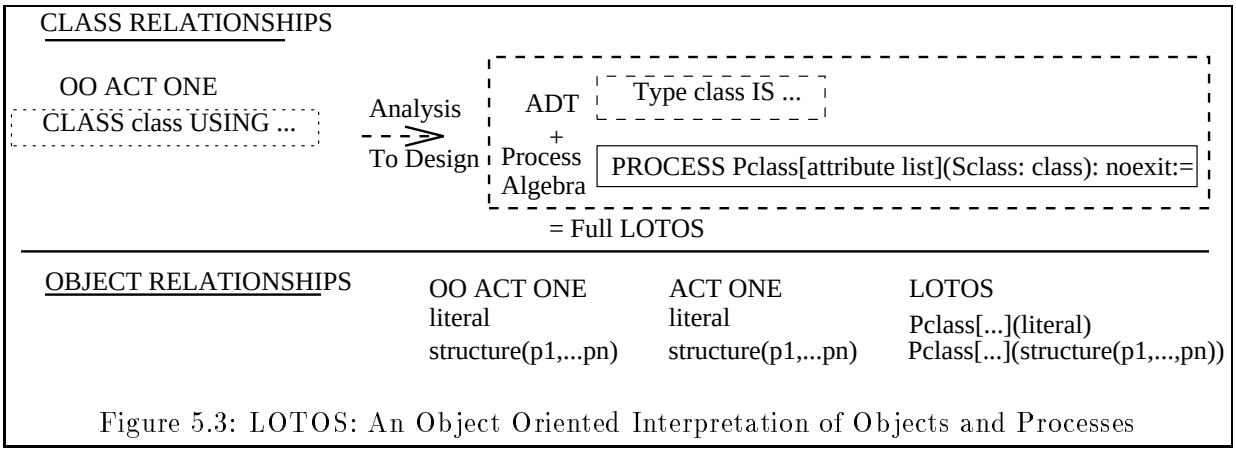


Figure 5.3: LOTOS: An Object Oriented Interpretation of Objects and Processes

event' of the form: `attribute?p1:P1?...?pn:Pn`, where there is an external attribute, `attribute`, defined to have class parameters `P1,...,Pn`. The result of an accessor or dual attribute is modelled as an 'output event' of the form: `accdual!result`, where `result` is a value of the appropriate ACT ONE sort. These correspondences are shown, for `Stack` behaviour, in figure 5.4.

	TRANSFORMER	DUAL/ACCESSOR
OO ACT ONE	empty.push = S(empty,0)	empty.pop = empty AND ~Nat
ACT ONE	.(push(empty,0)) = S(empty,0)	pop(empty) = dualStackNat(empty, unspecNat)
LOTOS	(PStack[push,pop](empty) |[push,pop]| (push!0; Q[push,pop])) = push!0; (Q[push,pop] |[push,pop]| PStack[push,pop](S(empty,0)))	(PStack[push,pop](empty) |[push,pop]| (pop; Q[push,pop]) = pop; (Q[push,pop] |[push,pop]| (PStack[push,pop](empty)|||(pop!unspecNat;exit)))

Figure 5.4: LOTOS: An Object Oriented Interpretation of Service Requests

5.5.4.4 Composition

The object oriented composition properties of the initial designs are contained in the ACT ONE part of the full LOTOS specification. The compositional properties can be derived from the ACT ONE code, or from examination of the OO ACT ONE specification (and associated diagrams). During the design process, expansion transformations (see *StExp* in 5.7.1, for example) facilitate the internal decomposition of a given unstructured process into a system of component processes running in parallel. The configuration and communication aspects of component interaction are made concrete by the design CPTs.

5.5.4.5 Subclassing In Design

An obvious question is whether the subclassing relationships in the requirements models are somehow maintained across the transformation. More formally, given OO ACT ONE classes, A and B say, such that $A \sqsubseteq B$ and a transformation function, T say, which maps OO ACT ONE classes to LOTOS processes, is there any relationship between processes $T(A)$ and $T(B)$? In OO ACT ONE, only two subclassing relations are defined, namely extension and specialisation. Consequently, it is necessary only to investigate how these two relationships are carried across the analysis to design translation.

A Design Extension

The `Stack` behaviour is extended below for the RPC communication model. The extension is a `size` attribute which returns the number of elements currently on the stack. The new class of behaviour, which offers this additional attribute, is named `XStack`. The RPC model of `XStack` behaviour is defined below.

```
process RPCXStack[push,pop,size](SXStack: XStack): noexit:=
(push? Nat1: Nat; RPCXStack[...](.(push(SXStack, Nat1))))
[]
(pop; pop! NatResult(pop(SXStack)); RPCXStack[...](.(pop(SXStack))))
[]
(size; size!NatResult(size(SXStack)); RPCXStack[...](SXStack))
endproc (* RPCStack *)
```

In the RPC model of communication, it is evident that corresponding[9] instances of processes `XStack` and `Stack` are not related by any of the standard testing equivalences [9, 10]: the `RPCStack` always deadlocks on event `size` whilst the `RPCXStack` does not. However, without going into formal details, the two process instances are related by a standard implementation relation, as defined in [16]: in a system containing an instance of `RPCStack`, the `RPCStack` can be replaced by the corresponding `RPCXStack` without the non-deadlocking behaviour of the system being compromised.

Consider the extension as it is carried across the *MakePar* mapping from analysis to design. The `ParXStack` code resulting from the *MakePar* mapping is given in Appx E1. The implementation relationship holds between `ParStack` process instances and `ParXStack` process instances.

A Design Specialisation

It is not possible to specialise the `Stack` behaviour since it is **nonpartitionable**. Consider instead a lift moving mechanism. The OO ACT ONE defining the `SMove` ('specialised move') and `Move` class interfaces, used in this example, is given below.

Consider the translation of this behaviour to an initial RPC LOTOS specification. (The same arguments apply for this model as for the Par model.) By definition, `Move` $\sqsubseteq$ `SMove` (since `Move` **spec** `SMove`). The LOTOS `RPCMove` and `RPCSMove` process classes are defined below.

[9] Corresponding instances of two processes are those instances with the same state representation.

```
CLASS SMove USING BOOL OPNS
LITERALS: up,down,stay
TRANSFORMERS: flip
ACCESSORS: goingup -> Bool
EQNS ...ENDCLASS (* SMove *)
CLASS Move SPECIALISES SMove TO OPNS LITERALS: up, down ENDCLASS (* Move *)
```

```
process RPCMove [flip, goingup ](SMove):noexit:=
(flip; RPCMove[...](.(flip(SMove))) []
(goingup; goingup!BoolResult(goingup(SMove)); RPCMove[...](.(goingup(SMove)))
endproc
process RPCSMove [flip, goingup ](SSMove):noexit:=
(flip; RPCSMove[...](.(flip(SSMove))) []
(goingup; goingup!BoolResult(goingup(SSMove)); RPCSMove[...](.(goingup(SSMove)))
endproc
```

The process algebra definitions for these two behaviours are identical, except in the naming of the processes and process parameters, and the typing of these parameters. It is immediate that corresponding instances of these processes are weak bisimulation equivalent (written $\sim$) as defined by [9]. In other words, `PMove[...](up)` $\sim$ `PSMove[...](up)` and `PMove[...](down)` $\sim$ `PSMove[...](down)`. The process instance `PMove[...](stay)` has no correspondences to any of the instances of `PMove`. This is precisely what is meant by **specialisation** in the object oriented semantic framework.

Definition: Class Relationships

The notion of a relationship between process instances is naturally extended to the notion of a set of relationships between sets of process instances. In this way, the notion of a class relationship can be developed in LOTOS. In LOTOS one says that process instances are related by some well defined relation. In an object oriented LOTOS, based on OO ACT ONE, this relation must be extended to parameterised process definitions, which correspond to classes. Given a relation R, between LOTOS behaviour expressions, a class relationship $ClassR$ is defined as:

`PROCESS PX` $ClassR$ `PROCESS PY` $\Leftrightarrow$
$\forall x$ such that x is a value expression of sort `X`, then `PX[...]`$(x)R$ `PY[...]`(x).

5.5.4.6 Polymorphism in Design

The RPC and Par models of communication, as presented above, do not incorporate the notion of polymorphism in the process algebra parts of the design. The polymorphic properties are defined in the ADT part of the designs but problems arise if polymorphism is not incorporated in the process algebra. For example, consider the `Stack` behaviour. The `Stack` is defined to accept `Nats` as input parameters of the `push` operation. However, if `Nat` is defined to have a subclass, `evenNat` say, the `ParStack` process cannot synchronise on a `push!evenNat1:evenNat` event, even though `Stack1.push(evenNat1)` is well defined in the requirements (in the OO ACT ONE and ACT ONE

models).

ACT ONE does not incorporate inclusion polymorphism semantics, but the ACT ONE class models (sorts) are specified to accept subclass parameter values through use of coercion and operation overloading. A similar approach must be taken when transferring the polymorphic requirements to full LOTOS. The RPC and Par communication models need to be expanded to cope with subclass input parameters. This is easily done: for example, the `ParStack` process contains the code fragment:

```
push!evenNat1; ParStack[...](.(push(SStack, evenNattoNat(evenNat1)))).
```

The inclusion of polymorphism properties in the process algebra part of the LOTOS designs is necessary, but its presentation can be ignored. Rather than including all the subclass parameter options in our full LOTOS design listings, the polymorphic behaviour is not presented. The design transformations presented in this thesis do not affect the subclassing hierarchy and so it is not, at present, necessary to consider the polymorphism properties when designing. The RPC and Par models are defined to specify the required polymorphic behaviour, even though it is not presented in the remaining sections of this chapter.

5.5.5 An Object Oriented Style of LOTOS Specification

An important aspect of the object oriented LOTOS designs is the specification of the *MakeRPC* and *MakePar* mappings, which define the *style* of the RPC and Par processes. Appendix E2 defines these two mappings. The RPC and Par processes are unstructured. They form the basic building blocks in the OO LOTOS *style* of specification. Three other basic building blocks are defined as the result of applying CPTs to these process definitions:

- ERPC ('expanded' RPC) processes result from applying a static expansion CPT (*StExp* in 5.7.1) to RPC processes. This produces a structured definition of the required behaviour in which the behaviour is composed from a number of RPC component process (in parallel) under the control of a centralised process.
- EPar ('expanded' Par) processes are a result of applying *StExp* to Par processes. This CPT results in a structured definition of the required behaviour in which the specification is composed from a number of Par component process under the control of a centralised process.
- Dist ('distributed') processes are a result of applying the *Dist* CPT (see 5.7.3) to EPar processes. This transforms a structured system of Par processes which have centralised control into a structured system of self-controlled (no pun intended) processes.

The OO ACT ONE *style* of specification is one in which the system is defined as a Par, RPC, EPar, ERPC, or Dist process. Further all the classes in the system which have been expanded to process form must also be represented in one of these five ways. In effect, the *style* of specification is defined by the initial communication models (RPC and Par) and the design transformations which can be applied to classes specified using these models. Consequently, the *style* is dynamic: when CPTs become well accepted design mechanisms, then the resulting process definitions will become well accepted design components.

In FOOD, the limited number of base *style* components can be usefully represented in diagrammatic form. The graphical notation is illustrated in figure 5.5

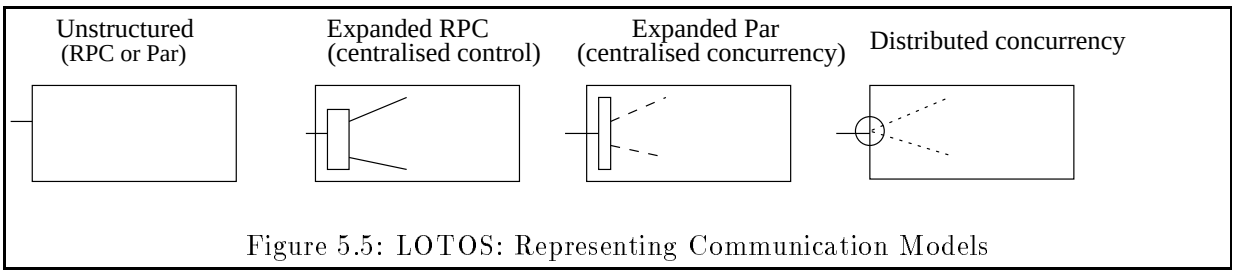


Figure 5.5: LOTOS: Representing Communication Models

5.6 Correctness Preserving Transformations (CPTs): Formalising Design

5.6.1 Introduction

This thesis examines the role CPTs can play in the process of design in general, and FOOD in particular. A transformation can be applied to a specification which reflects some architectural/implementation choice, without altering the external (observable) behaviour of the system. Such a design transformation is dependent on some nonstandard, though not necessarily informal, means of interpreting the internal details of the specification. The object oriented framework provides the basis for such an interpretation.

LOTOS, as a wide-spectrum language, can specify the properties of systems at various levels of abstraction. Design is the process which transforms an initially abstract (implementation independent) specification of system requirements into a final, more constructive, implementation oriented specification. An *ideal* LOTOS based software development environment should provide a comprehensive set of CPTs and a framework in which designers can apply these transformations to reflect design decisions. Such an ideal is a long way off. This thesis provides a small set of CPTs (in section 5.7) which are useful within our object oriented development method. These transformations are used to:

- Illustrate the CPT concept.
- Show the importance of matching design needs with CPTs.
- Highlight the difficulties involved in proving the correctness of the design transformations.
- Emphasise the power of a CPT-driven approach to design.

The small set of transformations proposed in this thesis do not constitute a design method. However, they do show how such a method could be constructed.

5.6.2 Concepts

5.6.2.1 Design Trajectory

A fundamental notion in this work is design trajectory: a sequence of steps which take a problem oriented specification of requirements to an implementation oriented specification of a possible solution. Each step changes the previous specification in some way. The important thing is that something must also be preserved along this trajectory: the 'correctness of the design'.

5.6.2.2 Design Verification

In theory, it is possible to verify the correctness of any given design step by mathematical means. In practise, the complete formal verification of most design steps is not possible because of combinatorial problems. In these cases, specifications are partly verified by simulation and testing.

This thesis has already identifed the advantages of simulation and testing with regard to analysis models. The same arguments are true for design models. The design approach advocated in this work does not restrict all design changes to be made through application of CPTs. Consequently, there may be a need for alternative verification methods. Two different types of LOTOS tools have been developed to help in this respect. Firstly, there are a wide range of simulation and automation tools (see [117, 10, 8], for example). Secondly, and more importantly, tools have been developed towards deriving tests from given LOTOS specifications (for example, [125] explains the theory behind a means of deriving canonical testers for LOTOS specifications). We do not examine any of these mechanisms, but we do recognise their value in this, and future LOTOS design methods. Rather, FOOD concentrates on a different approach to design verification, namely the application of CPTs.

5.6.2.3 Correctness Preserving Transformations

A different means of verifying a design is to perform only transformations (design changes) whose correctness has already been proven. Before examination of particular CPTs, a brief overview of the terminology is useful:

> A specification can be said to be *correct* if it fulfils some property. Assume a specification S, a transformation T and define $S' = T(S)$, i.e. S' is the result of applying T to S. T can be said to be correctness preserving with respect to the property P if $P(S) \Rightarrow P(S')$. In other words, the property P is preserved across the transformation T.

This type of formulation raises a number of interesting questions:

- What sort of properties can be usefully preserved?
- How can these properties be formalised?
- Over what domains should T operate?
- What is the difference between S and S' which makes T a useful transformation for applying during design?

- Can we specify appropriate transformations to correspond with decisions most commonly taken by designers in practice?

Before these questions are more rigorously examined, the concept of property is given a useful categorisation:

- **External Properties**
  External properties are those which can be observed through interaction with a system at its external interface (in LOTOS the external interface of a process is defined by its gate set). External properties, said to be purely functional, are fulfilled by a standard semantic interpretation of the specification. These properties are concerned with *what* the system does rather than *how* it does it.
- **Internal Properties**
  Internal properties are those which can be derived through examination of the text which specifies the system in question. They cannot be 'extracted' through interaction with the system interface alone. Formulation of these properties requires the definition of a non-standard interpretation of the specification. This interpretation is said to provide a **view** on the system.

This categorisation gives rise to the classification of two different types of CPT:

- **Structural CPTs**
  A structure CPT does not change the external properties of a system in any way. There are no ways of distinguishing the design before and after transformation through interaction with their external interfaces alone. Structure transformations change only internal aspects of the system.
- **Functional CPTs**
  A functional CPT changes the external properties of a system but guarantees some sort of conformance between the design before and after transformation. In other words, a functional transformation compromises some external properties but maintains others.

5.6.3 An Overview of CPTs in LOTOS

5.6.3.1 The CPT Problem

By differentiating between what should stay the same and what should be different, as the result of a design change, an elegant and formal statement of the requirements of a design step can be given as follows. Define:

- A specification S_1
- An implementation relation R
- A view function V, which has S_1 in its domain
- A view property P which is fulfilled by $V(S_1)$, i.e. $P(V(S_1))$ is true.
- A view property P' and a second view V' such that not$(P'(V'(S_1)))$

A **structured** design change corresponds to the specification of S_2, the next design, such that:

- $R(S_1, S_2)$, and R is a strong bisimulation equivalence[10].
- $P(V(S_2))$ and $P'(V'(S_2))$

In other words, S_2 maintains the external behaviour of S_1, maintains the view property P and adheres to a new view property P', which was not fulfilled by S_1. One could say that the reason for defining S_2 was the fulfilment of this new property.

A **functional** design change corresponds to the specification of S_2 such that: $R(S_1, S_2)$, and R is an implementation relationship which is not a strong bisimulation equivalence. In other words, a functional design changes the behaviour tree of the specification being transformed. The effect of such a change on view properties is specific to each design.

Some design steps can be defined as a mixture of the structure and functional approaches. In such instances, the behaviour tree is changed *and* view properties are maintained. Design CPTs provide a means of generating a suitable S_2 from any given S_1 such that the appropriate properties and relations are guaranteed.

5.6.3.2 The CPT Formulation

Section 6.6.3.1 focuses on the notion of a relation between two already specified design stages. It is useful to express the CPT problem in terms of transformations and constraints. We wish to discover a transformation T such that:

> Given any S_1 such that $P(V(S_1))$ and not$(P'(V'(S_1)))$, then:
> $R(S_1, T(S_1))$ and $P(V(T(S_1)))$ and $P'(V'(T(S_1))$

A Simple CPT Example

A LOTOS specification of a system is as a set of communicating processes. At this stage of development, the specification (design) has no multiway synchronisation. Between each pair of communicating processes there may be more than one synchronisation gate. We want a transformation which creates a new specification which conforms to the no multiway synchronisation constraint whilst guaranteeing the new property that there must be at most one gate shared between processes. Further, we require that the new design is a valid implementation of the old design.

This can be more formally specified, using the above notation, as follows:

- V, the view function, is defined to return a set of (process identifier $\times$ process identifer $\times$ gate identifier) triples, such that:

  $(p_1, p_2, g) \in V(S_i) \Leftrightarrow$
  p_1 and p_2 are defined to synchronise on gate g in LOTOS specification S_i.

- P the internal property is defined on S_i as:

  $P(V(S_i)) \Leftrightarrow$
  $((p, q, g), (r, s, g) \in V(S_i) \Rightarrow (((p = r) \Rightarrow (q = s)) or ((p = s) \Rightarrow (q = r)))$

[10]Strong bisimulation equivalence states that the behaviour trees offered by S_1 and S_2 are the same (even if the way in which they are specified is different).

- $V' = V$
- P' is defined on S_i as: $P'(V(S_i)) \Leftrightarrow$
  $((p,q,g1),(r,s,g2) \in V(S_i) \Rightarrow ((((p = r) and (q = s)) or ((p = s) and (q = r))) \Rightarrow g1 = g2)$

A transformation, T say, which fulfils these requirements is represented in diagramatic form in figure 5.6.

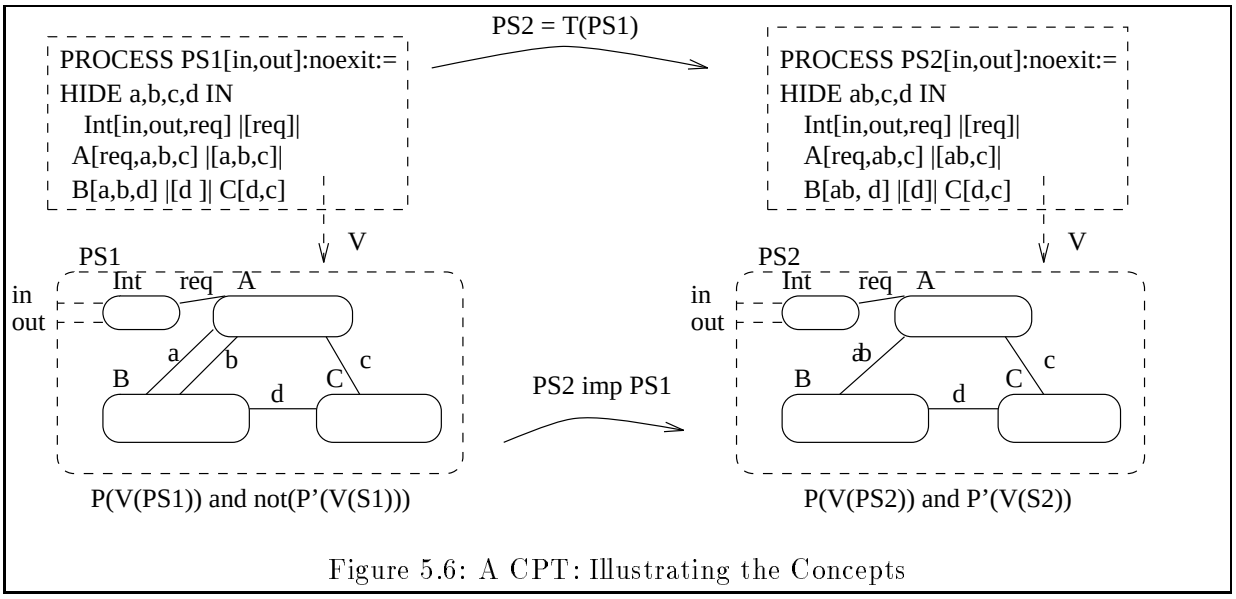


Figure 5.6: A CPT: Illustrating the Concepts

5.6.4 Graphical Views and Tools

5.6.4.1 The Need For Graphics

Designs and structures are often represented quite naturally in graphical notation. It is therefore desirable to be able to **view** a system of parallel communicating LOTOS processes in such a way that it is possible to extract a unique, meaningful, graphical representation. Then, design decisions can be represented as transformations on a **view**, with all the advantages of an underlying formal method.

Chapters 2,3 and 4 introduce graphical views of static and dynamic behaviour as specified in OO ACT ONE. This thesis recognises that the presentation of graphical views of process algebra specifications is much more difficult (the language constructs are much more complex) than that for producing ADT views. A recent thesis by Winstanley [126] examines the graphical presentation of static and dynamic properties of process algebra specifications. This work, however, does not consider the presentation of object oriented properties. It is important that a graphical notation for our object oriented LOTOS designs emphasises object oriented aspects. Graphs are useful to help customers (and analysts) to understand requirements models. This thesis supports the opinion that similar views would be useful to help designers communicate with each other, and the programmers. The formality underlying the graphical models used during object oriented requirements capture must also be evident during design. Graphics should not be open to interpretation.

5.6.4.2 The Need For Tool Support

The LOTOS object oriented designs lend themselves to the production of some sort of automated formal design environment:

- There are a limited number of standard class and object representations, each with well-defined properties.
- The CPTs can be easily automated.
- There is potential for developing a graphical syntax for the representation of the OO designs, based on OO ACT ONE graphics and the communication model notation.
- There is potential for the execution of such designs using existing tools

Certainly, the designers can use the object oriented analysis tools and models to understand the underlying functional behaviour. However, separate tools are needed to help the analysis of the communication and interaction properties of the object oriented designs. Graphical tools are particularly important for the representation of structural properties.

5.6.5 CPT Driven Design: Some Other Concerns

5.6.5.1 Problems With The Dichotomy of LOTOS

It is much easier to reason about a system when there is a 'conceptual consistency' in the way it is specified. Conceptual consistency depends on a reasoned approach to the way in which a problem is decomposed into its component parts. In LOTOS, a behaviour can be specified with different emphasis placed on the roles of the data typing and process algebra. However, there has been little research into how this division takes place; and more particularly, why some specifiers favour one 'half' of the language over the other. A consistent specification approach requires that the roles of each 'half' of the language is clearly defined at each stage of the development. In practice, specifications do not seem to have this consistency. The object oriented development strategy in this thesis makes a clear distinction between the fundamental behaviour, as defined by the ACT ONE part of the specification, and the communication, timing and architectural aspects, as specified in the process algebra part.

A more pressing problem with full LOTOS, with respect to formilising transformations, is that proof systems for data algebras are generally distinct from proof systems for process algebras. Combining two systems in one coherent transformation proof framework is very difficult. This thesis avoids the problem of proving correctness in two different formal frameworks by maintaining the ADT part throughout the whole design process.

5.6.5.2 Practicality must be the driving force.

The notion of basing a whole development method on a CPT system is very tempting. However, we believe that, although the area of formal design is amenable to CPT techniques, it is not possible to force all design changes to be done using CPTs. CPT research must be driven by the needs of designers. At the moment, designers are repeatedly making the same sort of structure decisions on

different problems. These types of decisions must be identified and then formilised within the CPT framework: designers will then be able to incorporate formal techniques within their work without needing to directly manipulate the LOTOS code. There is potential for automation of the underlying formal transformations. This thesis gives only a flavour of what is possible. The CPTs are defined only in an object oriented framework and it is clear that many more CPTs are needed. CPTs must be the main tool for formal object oriented design.

5.6.6 Object Oriented LOTOS CPTs and the Resulting Design Trajectory

The CPTs in this thesis are defined only on LOTOS specifications which have been derived from the OO ACT ONE specification, using the initial transformation to LOTOS. A sequence of CPTs can be applied to this initial specification to result in a correctness preserving design trajectory. Within this trajectory CPTs can be applied to the specification components (and the components of the components ...). It is not necessary for all transformations being applied to be pre-defined CPTs. In some cases, a CPT may be identified which may be of use in many different problem domains, but is not yet formulated for re-use. It is recommended that, in such cases, the designer attempt to formulate such a general CPT (if they can). However, if this is not possible (or desirable) then the designers must verify the particular transformation which they employ. The formal object oriented design trajectory, which forms the basis of our object oriented development method, is illustrated in figure 5.7.

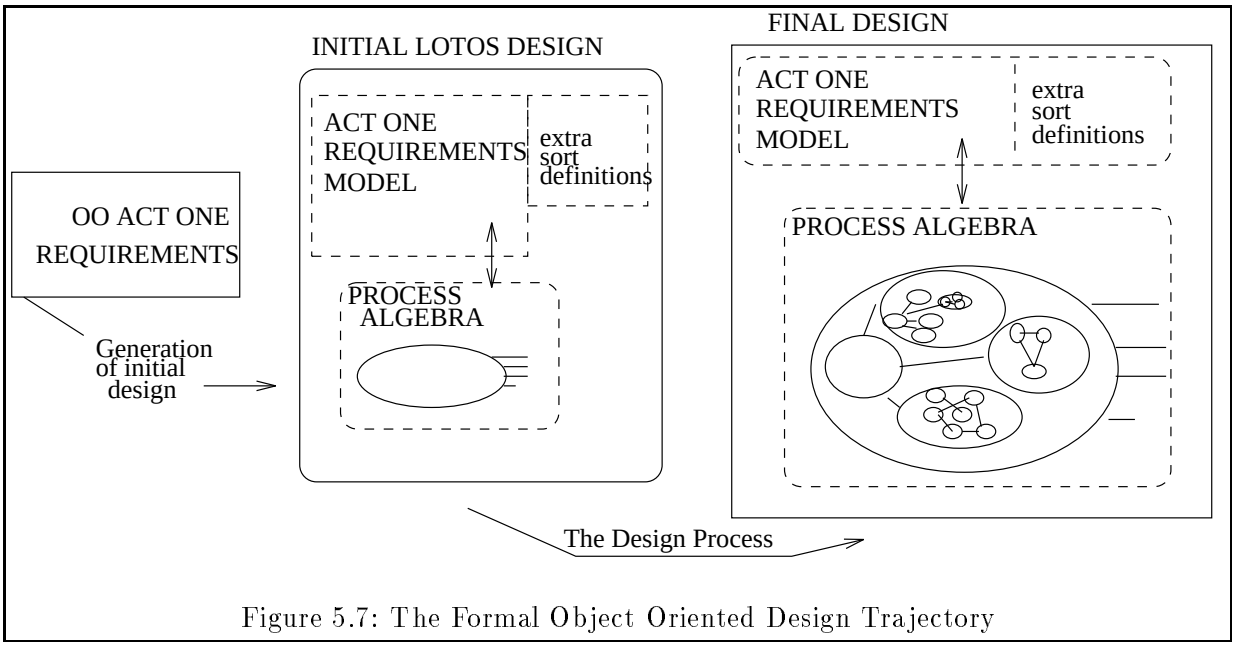


Figure 5.7: The Formal Object Oriented Design Trajectory

Notice that the ACT ONE code produced from the OO ACT ONE is maintained throughout development. There are two types of step in the design trajectory:

- A CPT-driven step, which does not need to be verified by the designer.

- A step, not achieved through application of a general CPT, which does need to be verified by the designer.

5.7 A Set of Object Oriented Design Decisions as CPTs

This section proposes five types of transformation which can be said to preserve the requirements as specified in the ACT ONE part of an initial LOTOS design. The transformations are used to illustrate the type of formal design which is possible within FOOD. The correctness of each transformation is argued informally: some rigorous reasoning is included, but it was beyond the scope of the thesis to prove the correctness of these transformations within the full LOTOS semantic framework. Future work must either:

- Define an object oriented design language whose semantics promotes the mathematical formulation of correctness and correctness preserving transformation.
- Address the problem of correctness formulation in full LOTOS, which arises out of the language being defined as a combination of an ADT and a process algebra. (The way in which our OO LOTOS specifications balance these parts of the specification makes this problem much more approachable than in the general case.)

The five transformations which we define are as follows:

- *StExp* ('static expansion') is defined on the domain of Par and RCP process classes which have a fixed structure and are defined *purely*. This transformation replicates the structure of a class in the requirements model in the specification of a system of component processes.
- *Comp* ('composition') is defined on the domain of statically expanded process classes. It provides a means of re-grouping a subset (or subsets) of the components of a system.
- *Dist* ('distribution') is defined on the domain of Par processes which have been statically expanded. It provides a means of removing a centralised control by distributing the control amongst the component processes. It relies heavily on the multi-way synchronisation mechanism in LOTOS.
- *Rend* ('remove nondeterminism') is a simple mechanism for the removal of nondeterminism in the requirements model.
- Finally, a general technique (not identified by a particular transformation) for the removal of parallelism is proposed.

The case study, in chapter 7, requires the designs to be targetted towards an Eiffel implementation. Consequently, since Eiffel has a procedural model of communication, the case study does not illustrate the *Dist* transformation. However, it does illustrate: the static expansion of *purely* defined classes with fixed structure, composition as a means of restructuring and the removal of nondeterminism.

5.7.1 Static Structure Expansion

Expansion is the term given to any transformation which expands out the process alegabra part of the OO LOTOS design with structure which is in the ADT part. The static expansion CPT (*StExp*) can be applied to Par and RPC class processes which have a fixed structure and *pure* specification. Informally, the static expansion transforms the unstructured class body process into a system of parallel processes. The static expansion of a `ParClass` process is shown in figure 5.8.

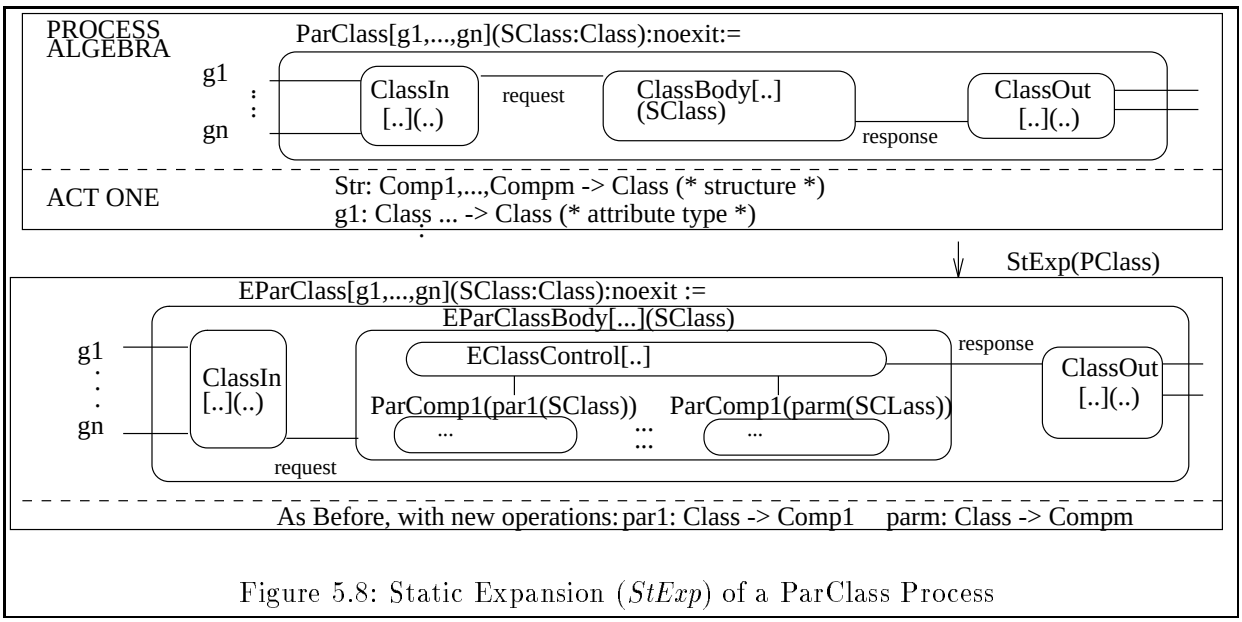


Figure 5.8: Static Expansion (*StExp*) of a ParClass Process

The static expansion of a `RPCClass` process is much simpler than that for the `ParClass`. It is illustrated in the design part of the case study (section 7.3). We do not report it here.

5.7.1.1 *StExp* Example: A System of Two Stacks

Consider a simple system of two stacks. The behaviour of the system is defined in the OO ACT ONE class `TwinStack`.

```
CLASS TwinStack USING Stack OPNS
STRUCTURES: TS< Stack, Stack >
TRANSFORMERS: push1<Nat>, push2<Nat>
DUALS: pop1 -> Nat, pop2 -> Nat
EQNS
TS(Stack1,Stack2).push1(Nat1) = TS(Stack1.push(Nat1), Stack2);
TS(Stack1,Stack2).push2(Nat1) = TS(Stack1, Stack2.push(Nat1));
TS(Stack1,Stack2).pop1 = TS(Stack1.pop, Stack2) AND Stack1..pop;
TS(Stack1,Stack2).pop2 = TS(Stack1, Stack2.pop) AND Stack2..pop
ENDCLASS (* TwinStack *)
```

The initial LOTOS design for this behaviour is generated using the *MakePar* mapping. This design is specified in process `ParTwinStack`. The process algebra specification for `PTwinStack` is given below.

```
process ParTwinStack[push1, push2, pop1, pop2 ](STwinStack):noexit:=
hide request, response in
TwinStackIn[ push1,push2,pop1,pop2,request ](0) |[ request ]|
TwinStackBody [ request, response ](STwinStack) |[ response ]|
TwinStackOut[ pop1, pop2, response ](0) where ...
```

The *StExp* CPT takes the `PTwinStack` process definition and produces a new process definition, named `EParTwinStack`. The CPT does this by leaving the definitions of the `TwinStackIn` and `TwinStackOut` processes alone whilst changing the `TwinStackBody` process specification. This is specified below.

```
process EParTwinStack[push1, push2, pop1, pop2 ](STwinStack):noexit:=
hide request, response in
TwinStackIn[ push1,push2,pop1,pop2,request ](0) |[ request ]|
EParTwinStackBody [ request, response ](STwinStack) |[ response ]|
TwinStackOut[ pop1, pop2, response ](0)
where ...
```

The new `EParTwinStackBody` is defined as a structured process in which there are three component (sub)processes:

- A control process, named `EParTwinStackControl` by convention, which, as its name suggests, controls the way in which the other components interact to produce the required behaviour.
- Two `ParStack` component processes: one for each component of the **structure** operation `TS`.

```
process EParTwinStackBody[request, response](STwinStack):noexit:=
hide Stack1push, Stack1pop, Stack2push, Stack2pop in
EParTwinStackControl[ Stack1push, Stack1pop, Stack2push, Stack2pop, request, response ](0)
|[ Stack1push, Stack1pop, Stack2push, Stack2pop ]|
( ParStack [ Stack1push, Stack1pop ](par1(STwinStack)) |||
ParStack [ Stack2push, Stack2pop ](par2(STwinStack)) )
where ...
```

There are a number of things worth noting about this specification, before details of the `EParTwinStackCon` process are considered.

- The hidden gates, namely `Stack1push, Stack1pop, Stack2push, Stack2pop`, have a 1-1 correspondence with the set of external attributes offered by the component classes of the `TwinStack`. These gates are identified by the component class name, followed by the parameter index of that class in the structure operation and finished by the attribute name.
- New sort operations, namely `par1` and `par2`, are used to return the individual parameter values of any given `TwinStack TS` structure representation. These new operations are generated by the *StExp* transformation, and added to the ADT part of the specification.
- The composition structure of the `TwinStack` has been expanded out in the process algebra part of the resulting design. This structure is still present in the ADT part, but it is now explicit in the communications model.

- The underlying functionality is contained in the ParStack components.

The *StExp* transformation is concerned mainly with the generation of a suitable Control process for any given statically structured class. ETwinStackControl illustrates how such a process is generated, for a simple behaviour.

```
process EParTwinStackControl[ Stack1push, Stack1pop, Stack2push, Stack2pop, request, response ]:
noexit:=
(request!push1?Nat1:Nat?ID:Nat;
Stack1push!Nat1; (EParTwinStackControl[...] ||| (response!push1!ID; exit))[]
(request!pop1?ID:Nat;
Stack1pop; Stack1pop?Result:Nat;
(EParTwinStackControl[...] ||| (response!pop1!Result!ID; exit))[]
(request!push2?Nat1:Nat?ID:Nat;
Stack2push!Nat1; (EParTwinStackControl[...] ||| (response!push2!ID; exit))[]
(request!pop2?ID:Nat;
Stack2pop; Stack2pop?Result:Nat;
(EParTwinStackControl[...] ||| (response!pop2!Result!ID; exit))
endproc (* EParTwinStackControl *)
```

The EParTwinStackControl specification is simple to generate because the two Stack components are not configured. The external attributes of the TwinStack are serviced by the Control 'passing them on' to the components, using the new internal gates.

Consider now extending the TwinStack behaviour with a swaptops transformer such that Stack1 and Stack2 are configured on swaptops. The swaptops attribute is more formally defined as:

```
TS(Stack1,Stack2).swaptops = TS((Stack1.pop).push(Stack2..pop), (Stack2.pop).push(Stack1..pop));
```

This attribute is translated into the EParTwinStackControl process by the inclusion of a new choice behaviour expression:

```
...[] (request!swaptops?ID:Nat;
(( Stack1pop; Stack1pop?Result1:Nat; exit)
|||
( Stack2pop; Stack2pop?Result2:Nat; exit ) )>>
(( Stack1push!Result2; exit) ||| ( Stack2push!Result1; exit) )>>
(EParTwinStackControl[...] ||| (response!swaptop!ID; exit)) ) ...
```

This more complex attribute gives a better flavour of how, in general, external attributes are translated by the *StExp* CPT. The resulting behaviour expression is made up of four parts:

- i) Accept the request and input parameters.
- ii) Perform internal accessor and dual operations for each dependent component.
- iii) Use the information gathered, if necessary, to perform internal state transitions (via external transformer requests).
- iv) Offer the response (with result in the case of a dual or accessor) in parallel with the original Control behaviour

When an external attribute depends on only one component then the translation is simplified by not having to use the ||| or >> operators. The `push1, pop1, push2` and `pop2` attributes, in `TwinStack`, are a good example of this.

5.7.1.2 The Correctness Of The Static Expansion

We are required to prove that `ETwinStack` is a **class implementation**[11] of `PTwinStack`. In other words, given `STwinStack`, a value of the `TwinStack` sort, then `ETwinStack[...](STwinStack)` **impl** `PTwinStack[...](STwinStack)`. The *StExp* relation between these two behaviours is illustrated in figure 5.9.

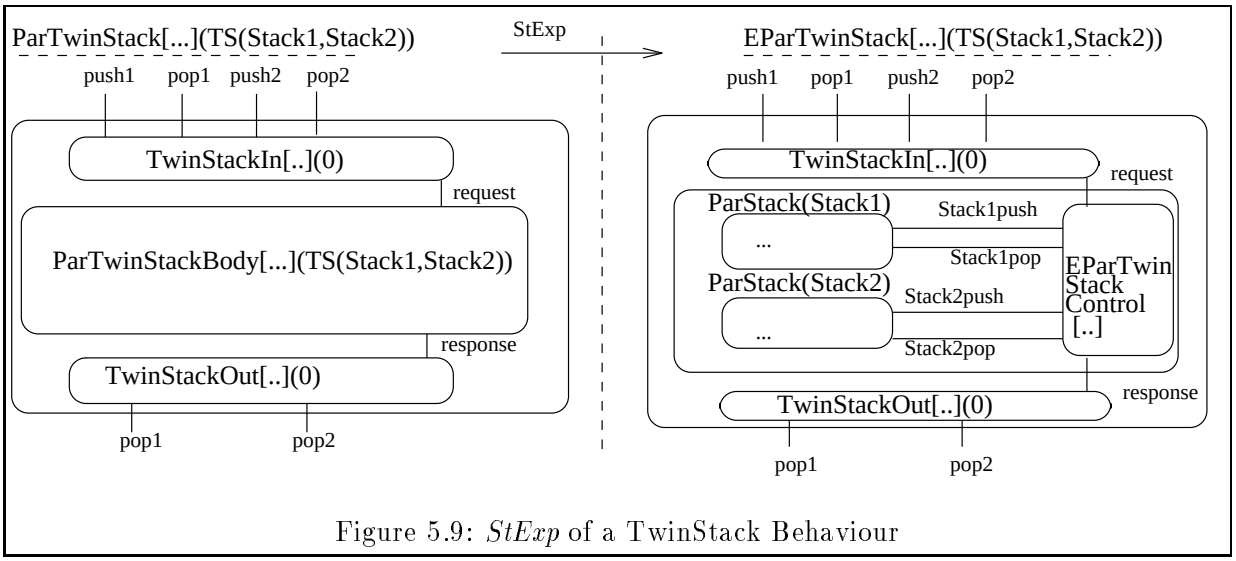


Figure 5.9: *StExp* of a TwinStack Behaviour

The two specifications have `TwinStackIn` and `TwinStackOut` components in common. The only difference is betwen the unstructured `ParTwinStackBody` and the structured `EParTwinStackBody` process instances. The `EParTwinStack` is an implementation of the corresponding `ParTwinStack` instance because of a simple property of the **implementation** relationship: any behaviour expression which contains an internal event can have that internal event transformed into a sequence of internal events (or vice-versa) without changing the external behaviour of the expression The *StExp* transformation changes internal requests (and responses) into sequences of internal events which model the passing on of the requests to the component processes, and responses back again.

The `TwinStackIn` and `TwinStackOut` processes, common to the design specification before and after the transformation, guarantee the external ordering of service requests and responses, no matter what changes are made to the internal sequence of events. Further, the use of the ADT specification to provide the underlying functionality guarantees comptibility between the behaviours offered. The transformation cannot introduce livelock or deadlock and so correctness is preserved.

[11] The **implementation** relationship is one which guarantees the preserving of the requirements in the original OO ACT ONE model.

5.7.1.3 Other Complexities To Be Addressed

The `TwinStack` example was chosen for its simplicity, and as such it does not address all the complexities of the transformation. These are as follows:

- **Preconditioned Equations in the Requirements Model.**
  Preconditioned equations are defined on structured classes as boolean expressions which depend on accessor[12] service replies from components. Preconditioned equations translate quite naturally into guarded expressions in the process algebra. Note that the completeness of the preconditioned equation (guaranteed by the `OTHERWISE` construct) means that no deadlocks can be introduced by the generation of guarded expressions in the new design specification. Further, no additional nondeterminism can arise from more than one guarded expression being true (across a choice of behaviours).
- **Invariants in the Requirements Model.**
  Invariants are realised by 'global preconditions' on every attribute in a class. Consequently, invariants are translated into 'global guards' in the process algebra. This can, unfortunately, lead to deadlocks when invariants are not proven, in the analysis stage, to be maintained throughout the lifetime of an object.

The complexities arising from the handling of preconditions and invaraints are not considered in any of the CPTs that follow.

5.7.2 Compositional Re-Structuring For Re-Use

There are two important aspects to restructuring for re-use: decomposition and composition. It is necessary to be able to decompose larger components into smaller ones so that the smaller components which have already existing implementations can be re-used. The static expansion transformation (*StExp*) provides a decomposition mechanism. It is also necessary to be able to compose smaller components into larger ones so that the implementation of the larger component can be re-used. It is this type of transformation which is considered in this section.

A simple solution to the re-structuring problem is to define a CPT which is the inverse of the *StExp* CPT. However, this is not general enough, since the designer may wish to combine only a subset of the component parts rather than all of them. Consider a `Class` which has five components. The designers wish to combine components 1,2 and 3, and components 4 and 5 to create new components (component1' and component2'). These new design components correspond to some already implemented behaviour which can be re-used directly. This restructuring is illustrated in figure 5.10.

The *Comp* CPT acts on any given statically expanded LOTOS specification. It is parameterised on a partitioning of the component set. In the diagram above, the partitioning is: $\{\{component1, component2, co$ $\{component4, component5\}\}$.

[12] In the requirements model preconditioned equations can be defined only on accessor attributes so that component state changes cannot arise from the evaluation of the precondition boolean expression.

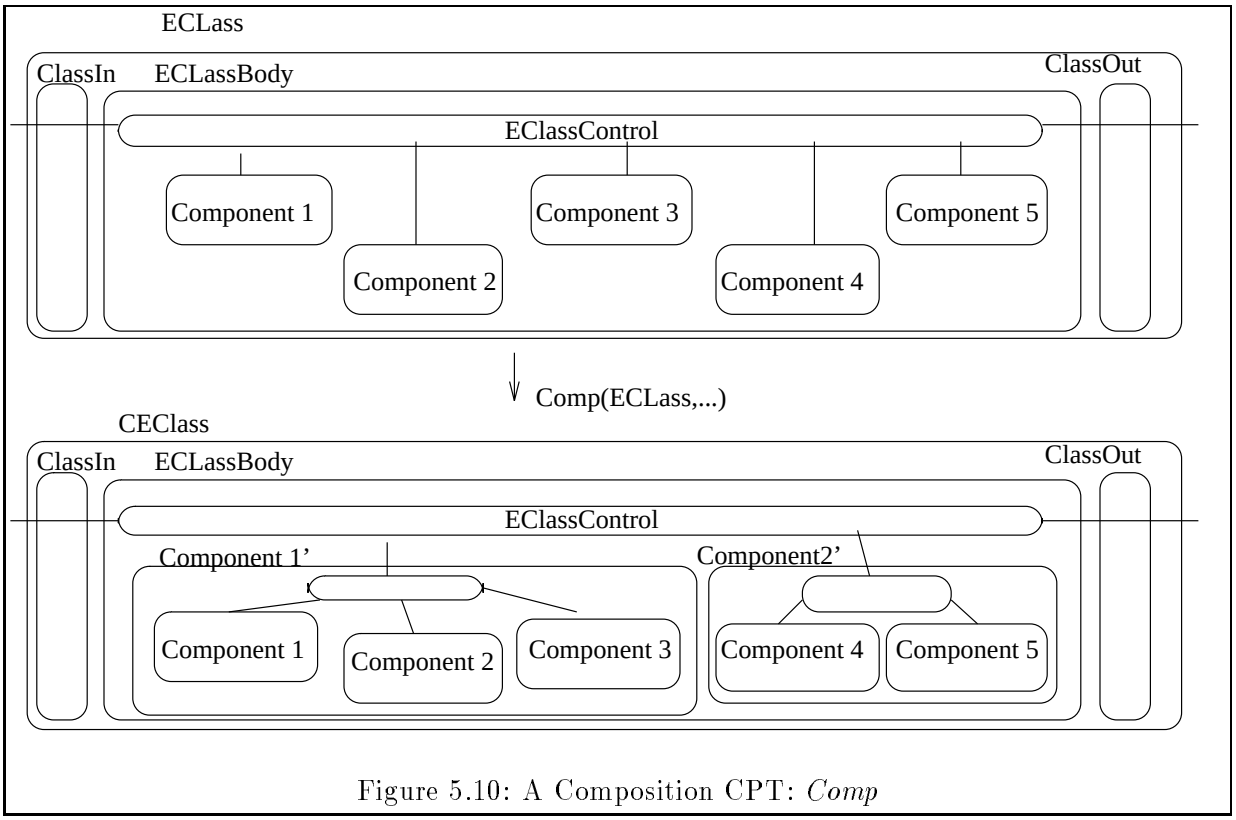


Figure 5.10: A Composition CPT: *Comp*

5.7.2.1 *Comp* CPT Example

Consider a system of two queues (of natural numbers) and a transformer The system accepts `Nat`s via the `on` attribute, transforms and then queues them up via the internal `trans` attribute, and outputs the results in their original order via the `off` attribute. This is more precisely specified by the OO ACT ONE `System` class definition, below.

```
CLASS System USING Queue, Transformer OPNS
STRUCTURES: SQQT < Queue, Queue, Transformer >
TRANSFORMERS: on<Nat>, trans (* internal *)
DUALS: off -> Nat
EQNS
SQQT(Queue1,Queue2,Transformer1).on(Nat1) = SQQT(Queue1.push(Nat1),Queue2,Transformer1);
SQQT(Queue1, Queue2, Transformer1).trans =
  SQQT(Queue1.pop, Queue2.push(Transformer1.tr(Queue1..pop)), Transformer1);
SQQT(Queue1,Queue2,Transformer1).off = SQQT(Queue1,Queue2.pop,Transformer1) AND Queue2..pop
ENDCLASS (* System*)
```

The `System` class structure diagram is represented in figure 5.11.

Static expansion of the `ParSystem` process results in the `EParSystem` process definition, as partially defined by the `EParSystemBody` process, below (the other parts of the `EParSystem` specification are not affected by the *Comp* transformation).

Now, the designers may be aware of a precoded component, `DoubleQ` say, which provides the

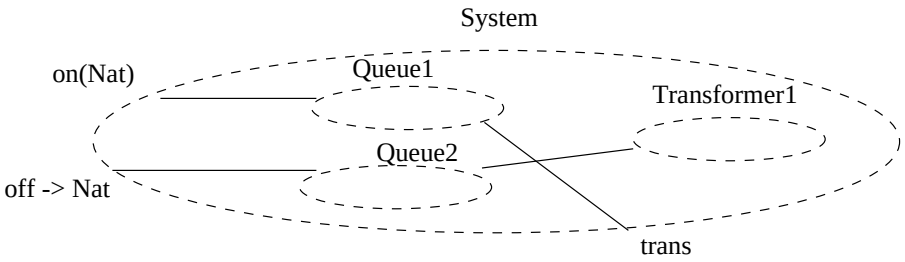


Figure 5.11: A Composition Example

```
process EParSystemBody[request, response] (SSystem:System):  noexit :=
hide Queue1push, Queue1pop, Queue2push, Queue2pop, Transformer1tr in
SystemControl[...] | [...] |
( ParQueue[ Queue1push, Queue1pop] (par1(SSystem)) |||
ParQueue[ Queue2push, Queue2pop] (par2(SSystem)) |||
ParTransformer[ Transformer1tr] (par3(SSystem)) )
where ...
endproc (* ESystemBody *)
```

functionality of two natural number queues which are linked in some unspecified way. Rather than having two distinct `Queue` components in the design, it is advantageous to combine them together into a single component. This can be done using the *Comp* CPT.

Comp(ESystem, $\{\{1,2\},\{3\}\}$) results in a new process specification which differs from the old process only in the specification of the `ClassBody`. The new process class body is named `CEClassBody`, in this case. The specification of `CESystemBody`, resulting from *Comp*(`Esystem`, $\{\{1,2\},\{3\}\}$) is given below.

```
process CESystemBody[request, response] (SSystem:System):  noexit :=
hide Queue1push, Queue1pop, Queue2push, Queue2pop, Transformer1tr in
SystemControl[...]
| [...] |(
CQueueQueue[Queue1push,Queue1pop,Queue2push,Queue2pop] (par1(SSystem), par2(SSystem))
||| ParTransformer[ Transformer1tr] (par3(SSystem))
)where
process CQueueQueue[Queue1push,Queue1pop,Queue2push,Queue2pop]
(Queue1:Queue,Queue2:Queue):noexit:=
ParQueue[ Queue1push, Queue1pop] (Queue1) ||| ParQueue[ Queue2push, Queue2pop] (Queue2)
endproc (* CQueueQueue *) ...endproc (* ESystemBody *)
```

The new `CQueueQueue` process can now be implemented using the pre-coded `DoubleQueue` component. For consistency, it is beneficial to be able to respecify the `CQueueQueue` process in standard `ParClass` form. Then, it can be transformed by any of the design CPTs. This standardisation requires the creation of a new ADT class, defined as a static structure with two `Queue` components. In other words, the ADT model of the new component is reverse engineered into a new OO ACT ONE class specification.

5.7.2.2 An Overview of the Correctness of *Comp*

The *Comp* CPT is doing nothing more than bracketing together sequences of interleaved operations and substituting internal events with sequences of internal events. Since the ||| operator is associative and the components being combined are interleaved processes, any bracketing of these processes can be done without altering the behaviour being specified.

5.7.2.3 Limitations of *Comp*

The *Comp* CPT works only on LOTOS specifications which have expanded static structure, with a centralised control component. Part of the job of an object oriented designer is to distribute the control aspects of a system among its component parts (see the *Dist* CPT in section 5.7.3). This distribution often means that the component processes are no longer interleaved, but must synchronise on shared gates. It is much more difficult to formulate a composition CPT for these types of distributed system specifications. This line of research is not examined in the design part of this thesis. Rather, during design we recommend that the expanded class specifications are compositionally restructured before the *Dist* CPT is applied.

5.7.3 Re-Structuring for Distributed Control

All structured LOTOS `EParCLass` processes have a centralised control to manage the way in which the component processes are used to provide the external functionality. The `EParClass` processes have a structure as shown in the left hand side of figure 5.12. The *Dist* CPT produces a `DistClass` structure, as shown on the right hand side of the same figure.

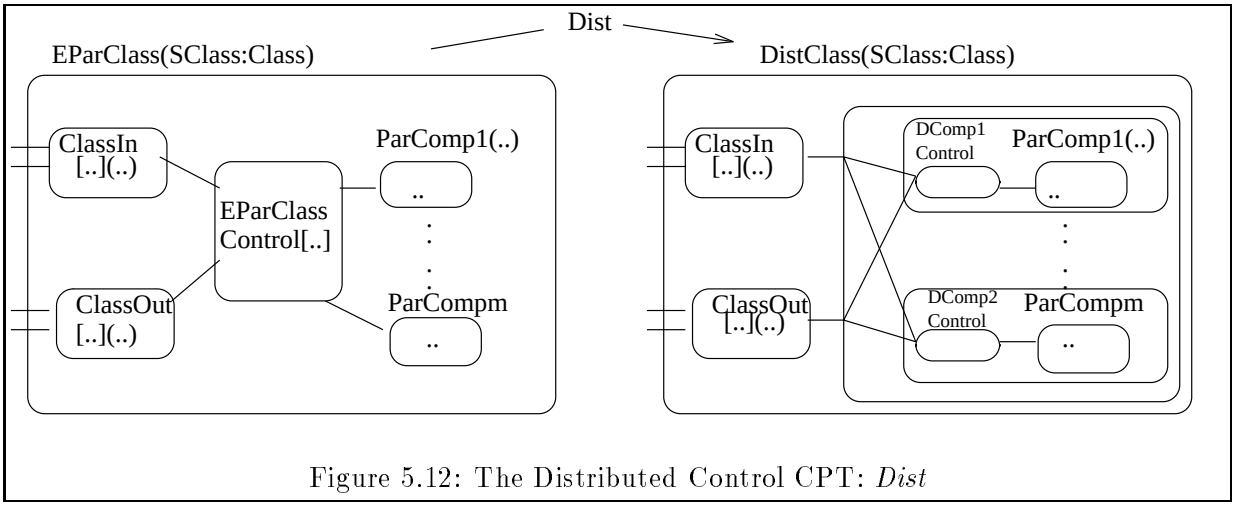


Figure 5.12: The Distributed Control CPT: *Dist*

Before the *Dist* CPT is applied, the `ClassBody` process has its concurrent processes under the control of the `ClassControl` process. In effect, there is a centralised process through which all requests and responses go. Object oriented designers may wish to remove this centralisation and distribute control in a decentralised fashion. There are potentially an infinite number of ways in which a designer could choose to do this. This section defines one CPT, namely *Dist*, which distributes the centralised

control of a `EClass` process amongst all of the `Class` component processes. The *Dist* CPT is defined on `EParClass` processes.

5.7.3.1 *Dist* of NonConfigured Structure: a TwinStack Example

Consider the `TwinStack` behaviour defined in 5.7.1. The *Dist* CPT applied to `EParTwinStack` produces the `DistTwinStack` process specification, below.

```
process DTwinStack[push1, push2, pop1, pop2 ](STwinStack):noexit:=
hide request, response in
TwinStackIn[ push1,push2,pop1,pop2,request ](0) |[ request ]|
DTwinStackBody [ request, response ](STwinStack:  TwinStack) |[ response ]|
TwinStackOut[ pop1, pop2, response ](0)
where
(* TwinStackIn and TwinStackOut are specified as before *)
process DTwinStackBody[request, response](STwinStack:  TwinStack):noexit:=
DStack1[request,response](par1(STwinStack))
|||
DStack2[request,response](par2(STwinStack))
where  ...
endproc (* DTwinStackBody *)
endproc (* DTwinStack *)
```

In the `EParTwinStack` class, the `ParStack` components are not configured. Consequently, there is no need for synchronisation between the `DStack1` and `DStack2` components of `DistTwinStackBody`. These two processes are interleaved to provide the required behaviour. Their specifications are given below.

```
process DStack1[request,response](SStack:Stack):noexit:=
hide Stack1push, Stack1pop in
ParStack[ Stack1push, Stack1pop ](SStack) |[ Stack1push, Stack1pop]|
DStack1Control[ request, response, Stack1push, Stack1pop ]
where
(* ParStack is specified in the normal way *)
process DStack1Control[ request, response, Stack1push, Stack1pop ] :noexit:=
(request!push1?Nat1:Nat?ID:Nat; Stack1push!Nat1;
(DStack1Control[...] ||| response!push1!ID; exit)) []
(request!pop1?Nat1:Nat?ID:Nat; Stack1pop; Stack1pop?Result:Nat;
(DStack1Control[...] ||| response!pop!Result!ID;exit))
endproc (* DStack1Control *) endproc (* DStack1 *)
process DStack2 ...(* Defined similarly to DStack1 *)
```

5.7.3.2 Distribution of Configured Structure: A TwinStack Extension Example

Consider an extension of the `TwinStack` behaviour in which the two components are configured by a `swaptops` attribute, which is defined as:

```
TS(Stack1,Stack2).swaptops =
TS((Stack1.pop).push(Stack2..pop), (Stack2.pop).push(Stack1..pop));
```

This results in the extended `TwinStackControl` process as defined earlier in this section. We must consider how such a configuration is transformed by the *Dist* CPT. Intuitively, there now must be some sort of internal gate shared by the component `Stacks`. The passing of information between components, which was originally done by the centralised control process, must now be done by the components.

In the *Dist* transformation, when processes $i_1, \ldots i_r$ are configured by an external attribute, then an internal configuration gate named `config`$i_1 \ldots i_r$ is defined in the resulting `ClassBody`. For example, since components 1 and 2 configure in the `TwinStack` (on the swaptops attribute) there is an internal gate defined as `config12` in the `DistTwinStackBody` process.

In the case where class components are configured, the *Dist* CPT produces a structured control process for every component. Each control is made up of `ServiceThese` and `IgnoreThese` components running in parallel, but not synchronised. The `ServiceThese` process controls the servicing of requests which depend on the particular component which it is controlling. The `IgnoreThese` process participates in all requests and responses which do not depend on the component to be fulfilled. This is necessary because *all* components must multi-way synchronise on request and response events. The `PStack` process, running in parallel with the control process, is defined in the normal way. Consequently, it can also be manipulated using CPTs whilst maintaing correctness. This is illustrated in the following code for process `DistTwinStack`.

The `DistTwinStack` example illustrates quite clearly how the configured servicing of a transformer `swaptops` is distributed amongst the two components. It is not clear, without further investigation, whether the distribution of control is as straightforward for dual (and accessor) attributes. In a 'well defined' OO ACT ONE specification of requirements, the result of a dual (and accessor) is always the result of a dual (or accessor) at one of the components of the structure. It is therefore quite natural in the distributed design for this one component to take responsibility for the result response. For example, consider a dual attribute `op` defined on a `TwinStack` as follows:

```
TS(Stack1, Stack2).op = TS((Stack1.pop).pop, Stack2.push(Stack1..pop)) AND T(Stack1.pop)..pop;
```

This results in additional fragments of LOTOS code in the distributed `TwinStack` design: the two `Control` processes, in each `Stack` component, are extended in the code below.

5.7.3.3 Overview of the *Dist* CPT Definition

The main complexity in the definition of *Dist* is the analysis of the configured attribute requirements. This analysis must identify whether attributes configure components. The parsing of the configured attribute requirements then splits the service into four parts:

- Performs accessors (and duals) on components which provide results for use in the internal requests in the remainder of the service. In the transformation, these result in a set of parallel internal events with 'data flow' modelled by the internal `config` event synchronisations.
- Dual events must be then be processed in order of nesting The ordering is maintained by the control parts of each component.

```
process DistTwinStackBody[request,response] (STwinStack:TwinStack):noexit:=
hide config12 in
DStack1[ request, response, config12](par1(STwinStack))
|[ request, response, config12 ]|
DStack2[ request, response, config12 ](par2(STwinStack)) where
process DStack1[ request, response, config12 ] (SStack:  Stack):noexit:=
hide Stack1push, Stack1pop in
PStack[ Stack1push, Stack1pop ](SStack) |[ Stack1push, Stack1pop ]|
DStack1Control [ request, response, Stack1push, Stack1pop ] where
(* PStack is defined in the normal way *)
process DStack1Control[...]:noexit:=
ServiceThese[request, response, Stack1push, Stack1pop] |||
IgnoreThese[ request, response ] where
process ServiceThese[ request, response, Stack1push, Stack1pop ]:noexit:=
(request!push1?Nat1:Nat?ID:Nat; Stack1push!Nat1; (ServiceThese[...] |||
response!push1!ID; exit)) []
(request!pop1?Nat1:Nat?ID:Nat; Stack1pop; Stack1pop?Result:Nat; (ServiceThese[...] |||
response!pop!Result!ID; exit))
(request!swaptops?ID:Nat; Stack1pop; Stack1pop?Result1:Nat; config12!Result1?Result2:Nat;
Stack1push!Result2; (ServiceThese[...] |||
response!swaptops!ID; exit))
endproc (* ServiceThese *)
process IgnoreThese[ request, response ]:noexit:=
(request!push2?Nat1:Nat?ID:Nat; IgnoreThese[...]) []
(request!pop2?Nat1:Nat?ID:Nat; IgnoreThese[...]) []
(response!push2!ID:Nat; IgnoreThese[...]) []
(response!pop2?Result:Nat?ID:Nat; IgnoreThese[...])
endproc (* IgnoreThese *) endproc (* DStack1Control*) endproc (* DStack1*)
process DStack2[ request, response, config12 ] (SStack:  Stack):noexit:=
(* Defined similarly to DStack1 *)
```

- The additional internal services that are required to achieve the correct global state of the system are treated separately as the penultimate part of the distributed service.
- Finally, the analysis identifies the component which is responsible for returning the result of the request (if it has a result). The reponse event is synchronised on by all components, but only one provides the result (the others accept any result value).

5.7.3.4 Overview of the Correctness of *Dist* On Configured Expanded Classes

As for the other structural CPTs, *Dist* does not change the external functionality of the system (class) being specified: it restructures the internal events (or sequences of events) which control the interaction between components of the system. Rather than having one central control process, the control is distributed amongst the components using multi-way synchronisation. Each component then decides which service requests it has to be involved in. Correctness is guaranteed because the `ServiceThese` and `IgnoreThese` processes guarantee the non-introduction of internal deadlock or livelock, the `TwinStackIn` and `TwinStackOut` processes maintain the same external communication

```
process DStack1Control[...]:noexit:=
ServiceThese[request, response, Stack1push, Stack1pop] |||
IgnoreThese[ request, response ] where
process ServiceThese[ request, response, Stack1push, Stack1pop ]:noexit:= ...
(request!op?ID:Nat; Stack1pop; Stack1pop?Result1:Nat; config12!Result1; Stack1pop?Result2:Nat;
(ServiceThese[...] |||
response!op!Result2!ID; exit)) ...
endproc (* ServiceThese *)
(* Process IgnoreThese defined as before *)
endproc (* DStack1Control*)
process DStack2Control[...]:noexit:=
ServiceThese[request, response, Stack1push, Stack1pop] |||
IgnoreThese[ request, response ] where
process ServiceThese[ request, response, Stack1push, Stack1pop ]:noexit:= ...
(request!op?ID:Nat; config12?Result1:Nat; Stack2push?Result1:Nat;
(ServiceThese[...] ||| response!op?Result2:Nat?ID:Nat; exit)) ...
endproc (* ServiceThese *)
(* Process IgnoreThese defined as before *)
endproc (* DStack2Control*)
```

interface, and the ACT ONE part of the design maintains the external functionality.

5.7.3.5 The Importance of the Distribution CPT *Dist*

The *Dist* CPT is the first step towards the formalisation of very complex systems of distributed objects (processes). It introduces the possibilty of modelling concurrent objects and shared objects at the high levels of design. This thesis is not concerned with the development of distributed software. However, the *Dist* CPT does illustrate how such work may be instigated in FOOD. There is much scope for developing a set of CPTs which can be applied to distributed `DistClass` processes.

5.7.4 Resolving Explicit NonDeterminism

This section addresses the need for designers to remove nondeterminsim in specifications. The CPT which we examine in this section is concerned with removing the nondeterminism due to `(* INTERNAL *)` transformations in the requirements model. One approach to removing nondeterminism is provided the by *Rend* ('remove nondeterminism') CPT.

5.7.4.1 Resolving Explicit NonDeterminism Using *Rend*: A CoinToss Example

Reconsider the simple `CoinToss` Class in section 4.3.4. The O-LSTSD is given, in figure 5.13, as a reminder of its behaviour.

The LOTOS `ParCoinToss` process, defined below, is the first high-level object oriented design of this behaviour[13].

[13]The removal of nondeterminism in the other types of object oriented LOTOS specifications is done similarly.

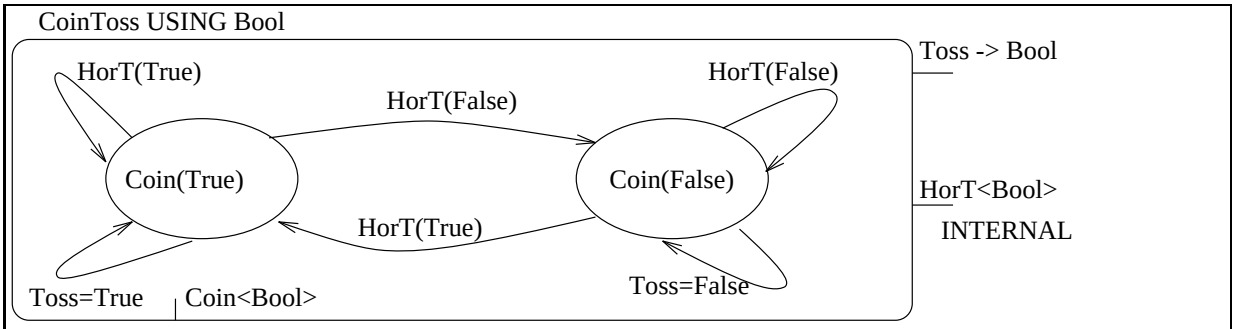


Figure 5.13: CoinToss: An Example of Nondeterministic Behaviour

```
process PCoinToss[ Toss ] (SCoinToss:  CoinToss):noexit:=
hide request, response, HorT in
CoinTossIn[ Toss, request, HorT ](0) |[ request ]|
CoinTossBody[ request, response ](SCoinToss) |[ response ]|
CoinTossOut[ Toss, response ](0) where ...
(* These processes are defined as if HorT was an external attribute.  *)
```

The `PCoinToss` specification says nothing about how, why or when the `HorT` internal transitions occur. Clearly, the designers must resolve this nondeterminism before implementation can begin. The *Rend* approach requires the specification of a new process which runs in parallel with the `CoinTossIn` process. This new process restricts when the internal transitions can take place. The *Rend* CPT takes a process, `DetCoinToss` say, as a parameter and produces a new `DetPCoinToss` specification, as defined below.

```
process DetPCoinToss[ Toss ] (SCoinToss:  CoinToss):noexit:=
hide request, response, HorT in
( DetCoinToss[ request, HorT ] |[ request, HorT ]|
CoinTossIn[ Toss, request, HorT ](0) )
|[ request ]|
CoinTossBody[ request, response ](SCoinToss) |[ response ]|
CoinTossOut[ Toss, response ](0) where ...
```

`DetCoinToss` can be any process specification which has a gate list [`request, HortT`] and is of type **noexit**. The correctness of the *Rend* transformation on `PCoinToss` depends on `DetCoinToss` fulfilling a simple property: at any stage in the behaviour of `DetCoinToss`, all external attribute request events (of the correct form) must be offered immediately or after a finite number of `HorT` events. This property guarantees the correctness of the *Rend* transformation. *Rend* places the responsibilty on the designers to prove that the required property is upheld. Fortunately, as the examples below show, this is often trivial.

5.7.4.2 A Set of More Deterministic Coin Tosses

In this section we define a set of `DetCoinToss` processes, each of which models a different way of removing some, or all, of the *HorT* nondeterminism in the `CoinToss` behaviour.

- I)

  ```
  DetCoinToss[...]:noexit:=
  request!Toss; HorT?Bool1:Bool; request!HorT!Bool1?Nat1:Nat; DetCoinToss[...]
  ```

  In this specification, `DetCoinToss` resolves only some of the nondeterminism by stating that after every `request!Toss` a state transition must take place before another `request!Toss` can be accepted. It says nothing about what state transitions occur between tosses.

- II)

  ```
  DetCoinToss[...]:noexit:=
  HorT!true; request!HorT!true?Nat1:Nat; request!Toss;
  HorT!false; request!HorT!false?Nat2:Nat; DetCoinToss[...]
  ```

  In this case, the designers resolve the nondeterminsim by stating that the coin tosses true and false, alternately.

- III)

  ```
  DetCoinToss[...]:noexit:=
  HorT!true; request!HorT!true?Nat1:Nat; request!Toss; DetCoinToss[...]
  ```

  In this case, the designers resolve the nondeterminsim by stating that the coin always tosses true.

These simple examples show the power in separating out the explicit resolution of internal transitions from the rest of the `DetClass` behaviour. The *Rend* CPT shows only one mechanism of resolving explicit nondeterminism in a controlled fashion. The domain of the *Rend* transformation is `ParClass` process specifications, but this can be easily extended. Like all the CPTs put forward in this thesis, *Rend* is used to show only that CPT based-design, in an object oriented LOTOS framework, has the potential for practical application.

5.7.5 Removing Parallelism

The object oriented LOTOS specifications, in this work, model concurrency using the parallel operators |||, || and | [...] |. Two processes combined by the parallel operator(s) can be said to be concurrent — of course the concurrency is just represented by an arbitrary interleaving of events. If the target implementation language supports concurrent entities then it is the job of the designers to match LOTOS processes to these entities. However, designers may wish to remove the parallelism when it is not supported at the implementation level, or if it is too fine-grain to warrant a mapping to separate implementation entities.

5.7.5.1 Removing Arbitrary Interleaving In Behaviour Expressions

The extension CPTs tend to produce design specifications in which parallelism models the arbitratry interleaving of communication events between a centralised control process and the component processes of which it requests services. For example, the `ETwinStack` services the `swaptops` attribute in the following way:

```
request!swaptops?ID:Nat;
((Stack2pop; Stack2pop?Result1:Nat; exit) ||| (Stack1pop; Stack1pop?Result2:Nat; exit))>> ...
```

The order in which the elements are popped off the two `Stack` components is not determined by the `ETwinStack` design. This leaves the designers some implementation freedom: the `TwinStack` may access the information concurrently or it may do it sequentially. The designer is free to remove the parallelism by changing the attribute definition. For example, a design decision to access `Stack1`, followed by `Stack2`, results in the following code:

```
request!swaptops?ID:Nat;
(Stack1pop; Stack1pop?Result1:Nat; Stack2pop; Stack2pop?Result2:Nat;)>> ...
```

Rather than attempting to specify a CPT which controls this type of design decision, we say that any behaviour expression in the object oriented designs written as `(P;exit) ||| (Q;exit)` can be transformed into `( P;Q; exit)` or `(Q;P; exit)` whilst preserving correctness.

Chapter 6

Object Oriented Program Derivation

This chapter examines how implementations can be derived from the formal object oriented LOTOS designs which arise from application of the methods defined in chapters 2 through to 5.

- **6.1: High-level Object Oriented Design as Input to Implementation**
  This section introduces implementation as an extension to design, and reviews a range of programming languages and environments which could be used to implement the object oriented LOTOS design specifications. It argues that, in general, executable languages can express three aspects of software specification: data structure, function (data transformation) and flow of control, and shows that different programming languages place different degrees of emphasis on each. With this in mind, the implementation of object oriented requirements using non-object oriented languages is first considered. Then, the advantages of working in an object oriented programming environment are put forward.
- **6.2: Object Oriented Programming: The Alternatives**
  Section 6.2 examines the different types of object oriented programming languages (and environments) which are currently available. It begins by defining the four main roles of object oriented programmers: interfacing with designers, writing code, producing documentation and testing. Different characteristics of object oriented languages are identifed and, based on these characteristics, a review of object oriented programming languages is given. Finally, Eiffel is chosen as the object oriented programming languages most suitable for implementing the formal object oriented LOTOS designs.
- **6.3: Translating Design To Implementation: Mapping Semantics**
  This section begins by reviewing the concept of targetted design: informality, in programming language semantics, is argued to make the targetting process more complex, and the future development of a programming language with formal semantics (based on the O-LSTS functional model, and a process algebra communication model) is recommended. The informal semantics of object oriented programming languages are reluctantly accepted as a necessary evil at this stage of the research. The remote procedure call communication model (RPC) is put forward as the best option when targetting design towards an Eiffel implementation.

- **6.4: Producing Eiffel from Procedural Object Oriented LOTOS Designs**
  Section 6.4 shows how Eiffel code can be developed from the formal object oriented designs, in order to meet the requirements. Initially, the implementation work of this thesis is placed within a set of reasonable bounds. It is not possible to examine all implementation issues and so restrictions are placed on the type of work which this thesis addresses. Then, an overview of the main implementation problem is given, namely matching value and reference semantics. The main body of this section gives a high-level view on the production of Eiffel code from formal designs: implementing object-based requirements, object-oriented requirements, using assertions and exceptions, and a potpourri of other relevant issues.
- **6.5: A Question Of Concurrency And Distribution**
  This section concludes this chapter by examining how the formal object oriented development process can be targetted towards concurrent or distributed implementations. It begins by stating the obvious advantages of concurrency and distribution in software systems, whilst re-stating the reasons for concentrating on a sequential implementation approach in this thesis. Alternative views of the relative merits of combining object oriented and concurrent models are given. The main problem for concurrent object oriented languages is argued to be that of scale. Using our object oriented design method is shown to provide a solution to the problem of complexity explosion when mapping objects to processes. Then, references to the conflicting requirements of object oriented and concurrent semantics are given. This section concludes by stating that the formal object oriented development approach, as advocated in this thesis, has the potential for being used to construct concurrent implementations.

6.1 High-level Object Oriented Design as Input to Implementation

In chapter 5, the importance of targetting a design towards a particular implementation language (environment) is stressed. Provided this is done appropriately, coding should then be a natural extension of the design process. Writing code should, in theory, be almost mechanical in nature, since the designers have done all the hard work. However, matching design specification semantics with different implementation language semantics is not always a simple task.

There are three orthogonal aspects to object oriented LOTOS designs:

- The communication model, i.e. the semantics of message passing (service requests and service fulfilment).
- The composition structure.
- The subclassing hierarchy and associated static and dynamic classification properties.

Each of these aspects must be mapped onto the implementation language. It is the designer's role to make this mapping as simple as possible.

The implementation process is made easier when the design semantics are close to the programming language semantics. Chapter 5 defines the object oriented design semantics in a way which

separates the functional, compositional and classification properties (in the ADT part) from the interaction, control and communication properties (in the process algebra part). Clearly, some languages will be better suited to implementing the formal designs than others. However, since, in general, programming languages provide the same computational power, the object oriented concepts can be mapped into non-object oriented language constructs.

6.1.1 An Overview of Programming Languages and Implementation Concerns

In general, executable (programming) languages can express properties with regard to three different aspects of a software system: data structure, data transformation (function) and flow of control. (This 3-dimensional categorisation is first commented on in chapter 2, when different aspects of analysis are considered.) Formal object oriented analysis concentrates on data structure and data transformation, whilst providing a service-request semantics which can be used to form the basis of a wide range of communication and control-flow models. The process algebra part of the formal object oriented designs make more concrete the flow-of-control aspects of the proposed solution to the requirements. Consequently, for implementation to be straightforward, it is necessary that the chosen programming language is rich in expression with respect to data structure, data transformation and control flow (data communication).

6.1.1.1 Data Structure and Data Relationships

The declarative composition and subclassing relations are fundamental aspects of the object oriented formal models. Programming languages, in general, have a declarative (non procedural) element, used to define problem specific data structures. Most programming languages provide a means of defining new data structures as groups (commonly called records or structures) of already defined data structures. These mechanisms provide an obvious means of modelling composition. However, most programming languages do not provide a mechanism for defining subclassing-like relationships between data structures. In such cases, a subclassing model must be built on top of the declarative constructs, using composition in some conventional way. Such a work-around solution is made even more complex by the polymorphic requirements inherent in formal designs.

6.1.1.2 Data Transformation (Function)

Data transformation is commonly provided by primitive operators (whose semantics are defined as part of the language), together with a means of constructing non-primitive operations (usually in the form of subprograms which can be called 'as-if' primitive). Procedural languages offer a similar degree of support for expressing data transformation properties. These transformation constructs provide a natural means of modelling the object oriented notion of attributes/methods. Non-procedural languages provide a different challenge to object oriented modellers.

6.1.1.3 Flow of Control

The different models of control flow evident in programming languages are categorised in figure 6.1. The main body of this chapter, namely sections 6.2 to 6.4, concentrates on non-concurrent implementation models. Section 6.5 examines the suitability of our formal object oriented development strategy with respect to the production of a concurrent implementation.

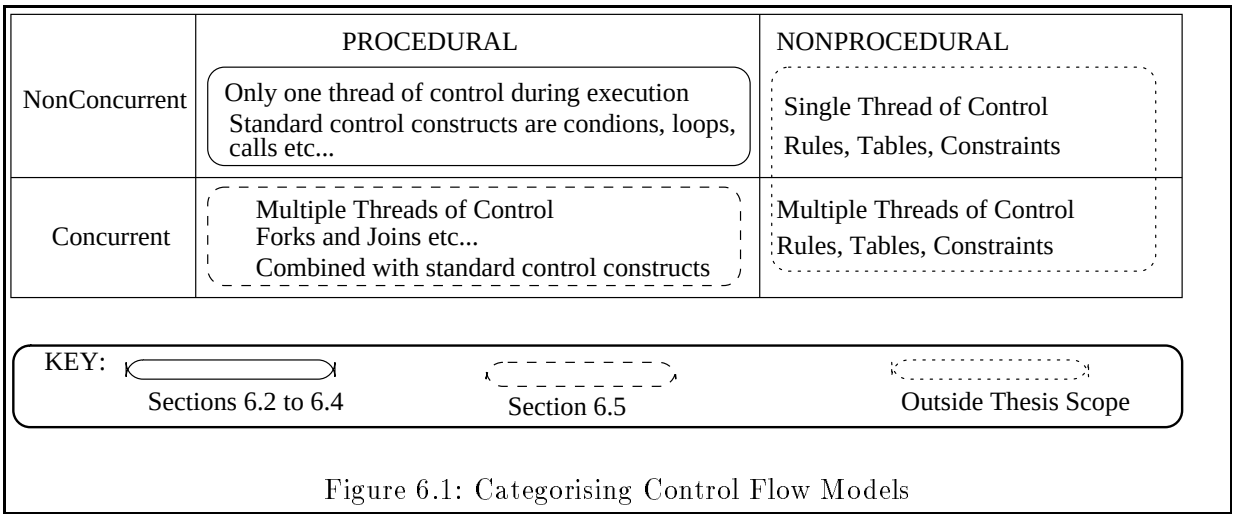


Figure 6.1: Categorising Control Flow Models

6.1.2 Implementation Outside an Object Oriented Framework

All languages (including object oriented programming languages) vary in their ability to support object oriented concepts. In particular, they differ in their ability to support the primitive concepts in the formal designs. Programming languages represent a compromise between: achieving a conceptual framework of understanding, being efficient and offering compatibility with other systems (and languages). Achieving a balance between these three requirements is, principally, what tempers the programming language semantics. Implementation is the process of matching these programming language semantics to the given design semantics. Using a non-object oriented language to implement object oriented requirements needs great care since there is no direct support, from the implementation environment, in maintaining the object oriented properties. However, it can be done. For example, Eiffel [84] is compiled into C, and the resulting C adheres to syntactic conventions which give it an object oriented flavour. By directly following these conventions, it is possible to produce a C implementation without using Eiffel. This implementation approach is very difficult without the type of support that the Eiffel programming environment provides. To give a flavour of how LOTOS object oriented designs can be implemented in non-object oriented programming languages, implementation in three different environments is considered:

- Using a purely functional programming language.
- Using an imperative programming language.
- Using a relational database language.

6.1.2.1 Functional Implementation

Functional and object oriented languages appear, at first glance, to have much in common: both place emphasis on the the notion of categorisation in the form of type and class, respectively. Much debate has arisen concerning the differences and similarites between the notions of type and class (and subtyping and subclassing). Section 2.5 provides an overview of the discussion and puts forward the stance of this thesis.

Given an ADT specification of object oriented requirements, as generated by the object oriented analysis, it is clear that it should be possible to directly implement it in a functional language: a simple mapping between sort and type, and operation and function would form the basis of such an implementation. However, as with the ACT ONE analysis model, a functional language which does not provide polymorphism will require additional work on the part of the programmer (or code generator) to fulfil the polymorphic requirements. Some functional programming languages provide polymorphic types, e.g. Miranda [111], but such polymorphism is ad-hoc rather than constrained. The work by Wadler and Blott [122] reviews the problems introduced by ad hoc polymorphism in functional languages. The research language Haskell [4] is an attempt to introduce object oriented properties into a purely functional programming framework, but this work is still incomplete. Perhaps the most interesting work in combining object oriented semantics with functional semantics is embodied in FOOPS (Functional Object Oriented Programming System) [60]. Unfortunately, the primitive definitions in the object oriented semantic framework used in this thesis do not correspond directly to the FOOPS notions. As such, although the mapping between the O-LSTS semantics and a functional semantics is an interesting task, a functionally based implementation was not carried out as part of this work.

6.1.2.2 Imperative Implementation

The expressive similarities of different imperative programming languages can be taken advantage of in the definition of a general algorithm for the imperative implementation of object oriented requirements. The key stages to such an algorithm are:

- **Stage 1: Model classes as data structures.**
  Classes are specified in the formal designs as structured processes. The composition structure is either: explicit in the decomposition of the process into a set of component processes, or implicit in the ACT ONE sort which parameterises the behaviour of class instances. In both cases, this structure can be directly translated into a record structure in an imperative implementation language. The fields of the record correspond to the components of the class. Variable record mechanisms (whether provided as primitive language constructs or defined by the programmer) can be used to model classes with different structures. Class literals can be simply implemented as enumerated types.
- **Stage 2: Model subclassing in data structures.**
  One approach to modelling subclassing is to flatten the class hierarchy structure: all the code for each attribute of a class is then defined in the particular class body. This can lead to

multiple copies of the same code, although in many imperative languages this duplication can be controlled using meta-language constructs like macros. A second approach is to model an inheritance facility using pointers to code bodies (which are shared among classes with common roots in the subclassing hierarchy). This second approach can be extended to model the dynamic binding of a service request to code at run-time. A third approach is to translate subclassing relationships into delegation relationships. Instead of a group of subclasses 'inheriting' code from a common superclass, the superclass behaviour being inherited can be defined in an extra component common to each of the subclasses. In this way, 'inherited' behaviour is provided by delegation. The programmers must choose which approach is best suited to their particular language.

- **Stage 3: Model polymorphism in data structures.**
  In our object oriented model, an object can be treated as if it was a member of any of its superclasses. This is polymorphism: the object is dynamically bound[1] to a particular class at run-time. The polymorphism must be controlled so that an object is only ever re-bound to a superclass of its current class. Polymorphism makes type checking complex. In imperative implementation languages, where the static type checking is comprehensive, it is necessary to model polymorphism using operation overloading and coercion. In languages with weak static typing, often the compiler does not check that typing properties are met, and so polymorphism is ad-hoc. This type of language can be used for the implementation of object oriented requirements, provided the typing requirements have been statically checked outside the domain of the programming language semantics.
- **Stage 4: Model attributes as functions.**
  Every attribute must be defined to have at least one argument, the implicit notion of *self*, i.e. the object being asked to service the attribute request. It is useful to define a convention that this argument is always the first one in the list of attribute operation parameters. Implementers must decide whether the other parameters should be passed by value or by reference. When passing parameter values as references, there may be side-effects if accessor or dual attributes are requested of the parameter. Consequently, for safety, it is better to pass the arguments as values. However, for efficiency reasons, it is often better to pass parameter values as references.

  A second concern when defining functions is the way in which they are named. Different programming languages have different syntactic restrictions placed on the naming of identifiers. It is important that a naming convention is found which, within these restrictions, can be used coherently and consistently. For example, a unique identity for each function can usually be generated by combining the class and attribute names in an appropriate fashion.
- **Stage 5: Model creation/initialisation routines.**
  Creating a system corresponds to instantiating a member of a class. Classes can be either:

  - **Purely static.**
    When a purely static class instance is created, the resulting system has a persistent struc-

[1] This is different from the notion of dynamically binding a service to code.

ture, i.e. servicing an arbitrary sequence of external attribute requests does not change the structured representation of the initial object. Further, in a purely static class, the components of the class are required to be purely static instances.

- **Impurely static.**
  As with a purely static class, the instances have persistent structure but, in the impure case, class components are not required to be purely static.
- **Dynamic.**
  A dynamic class instance, defined as a particular set of component objects, may be transformed into a different set of components by fulfilling some sequence of service requests.

Except in the purely static case, it will be necessary to be able to create and destroy system component objects (subsystems) during the lifetime of the system. These subsystems can, when the use of references is strictly controlled, be implemented as stack-based variables. The imperative language compiler can then automatically cope with memory allocation and deallocation.

- **Stage 6: Model encapsulation.**
  Object oriented semantics require encapsulation of an object so that access to its state must be done through its external interface. This requirement is not standard in imperative programming languages. Module-like constructs provide encapsulation in some languages, but there are difficulties in defining a correspondence between objects and modules, especially in dynamically structured systems. Furthermore, confusion can arise when systems have multiple instances of the same module. It is better to enforce a convention that all access to structured data (in record form) must be through the external attribute functions.
- **Stage 7: Model concurrency, or lack of it.**
  The object oriented LOTOS design models are easier to implement if the concurrency (modelled using the parallel operators) is removed. In this case, message passing (service request/service response) events can be modelled imperatively using remote procedure calls. Section 6.5 examines the issues which arise when the final design has concurrent aspects which are intended to be carried through to concurrent implementation language constructs.

These seven stages provide the basis for implementing object oriented requirements in an imperative language. Of course, this mapping of object oriented requirements is not the whole story for the implementers. They must also consider coding a user interface, fulfilling non-functional requirements, documentation, testing, etc ...(see 6.2). However, these aspects can be done *as-if* the implementation language was object oriented, provided the seven stages above are complete.

6.1.2.3 Implementation Using a Relational DataBase

Before examining implementation using object oriented programming languages, a final, less obvious, alternative is briefly considered: using a relational database.

When the object oriented requirements place emphasis on the persistence of data, i.e. data that exists beyond the lifetime of a single program execution, then a permanent data store is required.

Further, often the functional requirements of such systems are concerned with information retrieval and update: controlled access to different fields of data in a large data structure. Given such a requirements model, it is natural to think of implementing the system as a data base. Fortunately, high-level database languages exist to provide the core behavioural framework onto which particular data structure, and associated functionality, can be built. Relational database languages (see [23, 74] for example) have the potential to provide a sound basis on which to implement complex software systems of persistent objects.

6.1.3 Implementation in an Object Oriented Environment: The Advantages

Computational power is not an issue when choosing one implementation language over another, since programming languages can, in general compute whatever is computable. Section 6.1.2 gives an overview of how object oriented constructs can be modelled (and therefore implemented) in non-object oriented programming languages. In such an approach, the imperative language is used to construct a model of the object oriented semantics. There are inherent difficulties when implementing on top of such a model:

- The mapping between the object oriented primitives, in the LOTOS designs, and the final implementation language primitives is more complex than necessary.
- The imperative language does not provide error protection facilites, in the form of type checking, which can automatically check the object oriented implementation to guarantee it fulfils the complex correctness properties associated with a polymorphic language.
- The testing of the implementation becomes more complex since it is necessary to test both the functional requirements and the correct modelling of object oriented primitives.

One of the advantages of using an object oriented programming language is the consistent framework of conceptualisation between analysis, design and implementation[2]. Unfortunately, although the primitive concepts are common, the underlying semantics of the primitive concepts is not standard. Consequently, there is still a need to model the object oriented requirements primitives, as specified in our LOTOS designs, onto an object oriented implementation language. However, in most cases, the object oriented programming language semantics are closer to our design semantics than for non-object oriented programming languages, and the mapping is therefore much simpler. In particular, many of the mapping steps needed for imperative implementation (see 6.1.2.2) are unnecessary when using an object oriented programming language.

6.2 Object Oriented Programming (OOP): The Alternatives

6.2.1 The Roles of Object Oriented Programmers

Object oriented programmers have four main roles: interfacing with designers, coding, documenting and testing.

[2]Chapter 2 examines all the advantages of working in an object oriented framework, many of which are related to the conceptual consistency.

6.2.1.1 Interfacing With Designers

The role of designers is to target the requirements towards a particular implementation architecture. This targetting is at three different levels:

- Matching the object oriented semantics in the design with the programming language semantics, particularly with regard to the dynamic classification and communication properties.
- Matching the compositional structure of the design to resources in the implementation language. In particular, this means re-using already coded design/implementation components.
- Ensuring that non-functional requirements can be met by the chosen implementation environment.

It is advised that implementers help designers to make appropriate design decisions.

6.2.1.2 Coding

After object oriented analysis and design, the implementers may still have much to do:

- Code new classes and make these available for re-use (in some sort of package facility).
- Place new design classes into the class hierarchy, if not already done during design.
- Identify new generic classes and define these for re-use.
- Fulfil the non-functional requirements.
- Match the static and dynamic typing requirements to the programming language.
- Provide a user interface to the system: define a means of representing system state, a means of dynamically interacting with the system, and a way of storing and retrieving previous systems.
- Resolve the unspecified behaviour associated with exceptions, which was not dealt with during design.
- Resolve implementation freedom.

These tasks are clearly inter-related in a complex way. This thesis is not an examination of object oriented programming techniques and as such we do not examine the programming process in great detail.

6.2.1.3 Document The Implementation With Respect To Design

The formal design forms the basis of the code documentation. Each implementation class has an associated design component. The OO ACT ONE specification (defined by the ACT ONE code) for each sort acts as a good statement of functional and structural properties. As such, we recommend that it be included in the code (in the form of a comment). The process algebra specification of the communication model can be included when its requirements are complex: when a consistent RPC model is enforced it is not necessary to include the communication information.

Another important role of the documentation is to comment on differences between design and implementation. For example, when sharing is used for efficiency, or concrete state does not match

abstract state. Further, documentation must deal with re-use issues: where predefined code has come from, and where re-usable components are stored for future use. Documentation must also deal with testing and user manuals[3].

6.2.1.4 Testing

The formal approach advocated in this thesis cannot guarantee that the resulting implementation provides the required behaviour since there are two informal steps: customer communication of requirements[4] and implementation. Implementation is said to be informal because there is no formal mapping between the semantics of the LOTOS designs and the semantics of the resulting executable code. However, the formal analysis and design stages do guarantee that the requirements model is fulfilled by the final design, and this final design is unambiguous. Further, the object oriented framework aids understanding of these formal models. As such, it helps to cover the informality gap at each end of the development process. Testing is the process by which implementers bridge the gaps at their end of development.

Code is tested against the final object oriented design (the initial requirements, implicit in the design, have already been validated by the customer). The structure in the design matches structure in the implementation, to a great extent, and consequently the testing process can be incremental.

6.2.2 Characterisation of OOP Languages

Object oriented programming languages vary in their support of object oriented concepts. The object oriented semantic framework, defined in chapter 3, is the basis upon which we evaluate the suitablility of languages for implementing formal object oriented designs in LOTOS. Object oriented characteristics are categorised into three groups:

- **Essential**
  These characteristics are the minimum requirements for a language to be considered suitable for implementation.
- **Important**
  It is important that these characteristics are evident in the chosen programming language if the formal object oriented development method is to progress past the research stage, and gain initial acceptance in industry.
- **Beneficial**
  Beneficial characteristics are those which could eventually positively influence the widescale adoption of formal object oriented development within industry.

It is not always clear whether or not an implementation language (or environment) exhibits a particular characteristic. When a distinction is necessary, the following categorisation is useful:

[3] It is beyond the scope of this thesis to examine the production of customer documentation.

[4] It is common for the customer to validate the requirements model as being correct even when it does not exactly represent their needs. In an ideal environment, the formal requirements act as a contract between customer and software developer so that there is a level of customer liability.

- **Directly Supported**
  Directly supported characteristics are provided by the language primitives, but not necessarily enforced.
- **Supported**
  Supported characteristic are provided as elements of the language libraries, or can be easily coded as such.
- **Unsupported**
  Unsupported characteristics can be modelled by the language, but there is no language or library support.

6.2.2.1 Essential Characteristics

We regard the following characteristics to be essential in a programming language which is to be used for implementing the object oriented LOTOS designs:

- **Classification**
  All object oriented languages provide a means of defining classes of behaviour. In most cases, these classes are defined to have state attributes[5]. Objects are references to particular instances of a class, in which the state attributes have been set to particular values.
- **Encapsulation**
  Object state must be encapsulated behind an interface. In some languages, the state attributes cannot be accessed directly. In others, attributes must be declared *private* if direct access is to be prohibited. Unfortunately, some languages facilitate the declaration of state attributes as *private*, but do not enforce the privacy (see Smalltalk [58, 57], for example).
- **Composition**
  All object oriented languages facilitate a form of composition, usually by allowing state attributes to be defined as objects. An object can then be said to be composed from its state attribute values. This simple notion of composition is complicated when state attributes are defined as references to shared objects. Sharing is an efficiency matter which is not necessary for correct implementation of the object oriented designs.
- **Subclassing**
  All object oriented languages offer a subclassing mechanism. This mechanism is essential for polymorphic properties to be offered in a controlled manner. Unfortunately, object oriented programming languages provide subclassing in the form of inheritance, which performs two distinct roles: it defines the class relationships in the system and defines how these relationships can be used to implement the efficient binding of 'shared code' to a service request. In this thesis, the subclassing relationships are essential to provide inclusion polymorphism, whilst the code sharing aspects are secondary to this main issue.

[5]This notion of attribute is different from our well defined notion of attribute (as part of an object's interface), but more of this difference later.

It is also essential that the subclassing is supported by a 'multiple inheritance' mechanism, since we require that a class can be defined as a subclass of different superclasses which themselves are not related by a subclassing relationship. (Unfortunately, the way in which object oriented programming languages cope with conflicts in multiply inherited attributes is not consistent.)

- **Substitution Polymorphism**
  An object which is typed to be a member of a class, C say, must be acceptable as a member of any of the superclasses of C (this is the well accepted notion of substitutability). This type of requirement can be met by any untyped (ad-hoc polymorphic) language. However, this option is ruled out by the next essential characteristic: strong typing.
- **Strong Typing**
  When each variable in a system is known merely to be an object, of some unspecified sort, this is known as weak typing. Contrastingly, in strongly typed object oriented languages, every variable is precisely defined as belonging to a particular class. Strong typing is essential, in our opinion, because it provides facility for actively supporting the implementation of correct code. Ideally, type correctness in the implementation language is guaranteed by type correctness in the LOTOS design. However, in practice, implementations have typing aspects which are not directly checked by earlier development stages.

6.2.2.2 Important Characteristics

The characteristics which we consider important, but not essential are:

- **(Incremental) Compilation**
  It is important that the implementation code can be compiled into machine code. This requirement is purely an efficiency and portability concern. Incremental compilation is an additional advantage because it leads to the generation of autonomous re-usable implementation components.
- **Genericity**
  Genericity is not a subclassing mechanism, but it is a powerful technique for defining parameterised behaviour. Genericity improves understandibilty (by highlighting common structures) and encourages re-use.
- **Comprehensive Class Libraries**
  Most object oriented programming languages include a library of standard classes for general purpose data structures, file handling, user-interfacing, graphics, mathematics, etc .... Without these class libraries, object oriented programming is very difficult.

6.2.2.3 Beneficial Characteristics

It is beneficial for the following characteristics to be offered by the chosen implementation language, but not essential at this early research stage.

- **Assertions**
  Assertions can improve the mapping between requirements and implementation. Provided that the object oriented design assertions (defined as boolean expressions) can be expressed directly in the implementation language, there can be automatic checks, made during execution, that the requirements are fulfilled. Thus assertions (of some sort) in the implementation language can improve the testing process. A second consideration is that assertions improve the understandability of the code. Consequently, it is advised that assertions are placed in the code as comments, even when no mechanism exists for making the checks during execution.
- **Garbage Collection**
  Dynamic object oriented systems require the production and destruction of component objects during execution. Garbage collection is an important memory management facility which frees unreachable object space for future use. Some object oriented programming languages do not provide automatic garbage collection, but expect programmers to explicitly deallocate memory when an object is no longer needed.
- **Wide Acceptance (in industry)**
  It is important that we target our designs towards implementation languages which have a wide acceptance (industrial as well as academic). Widely used languages offer continual support via published work and second-hand user experience.
- **Packaging**
  A class is not an ideal fundamental building block for re-use. In many cases it is beneficial to be able to re-use groups of related classes (a package). Packages can help to control visibility between classes. Object oriented programming languages often require unique class identifiers. This is counter-productive to the independent production of compatible re-usable classes. Packaging can provide a means of defining name-space domains to avoid this problem.
- **Concurrent Constructs**
  Concurrent constructs have the potential to improve efficiency, increase resource utilisation and more naturally model the real world requirements of highly parallel systems. Concurrent constructs free designers from having to target the designs towards the constraining non-concurrent semantics which dominate programming languages at the moment. Concurrency also improves the extendibility of the system. Section 6.5 examines the issue of concurrency in more detail.
- **Tool Support**
  Software development tools (for example, debuggers, browsers, interpreters and syntax directed editors) have the potential to improve productivity. Also, they can improve the chances of the code meeting customer requirements. Tool support is particularly important in an object oriented implementation environment [57].
- **Persistency Support**
  A permanent data store is required by a large number of software systems. A persistency mechanism can simplify the implementation of a data store, and consequently make the code easier to understand.

- **Purity**
  Object oriented programming languages are categorised as being pure or hybrid. Pure languages are those which do not provide language constructs whose roles are outside the object oriented paradigm. Hybrid languages, being extensions of non-object oriented languages, provide language constructs whose roles are not necessarily object oriented. The problem with many hybrid languages is that object oriented principles are not enforced. Further, the non-object oriented constructs can be used to violate the object oriented requirements. Purity also makes the implementation code much more consistent. In general, consistency implies coherency. (Smalltalk, by enforcing the consistent notion that *everything* is an object, is an excellent counter example to this claim.)

6.2.2.4 A Note On The Importance Of Semantics

A separate problem occurs when characterising programming languages if their semantics for particular characteristics do not match the formal semantics in the formal designs. Eiffel and C++ provide two interesting examples of this:

- Eiffel offers a subclassing mechanism (inheritance), but this does not fulfil the contravariance requirement in the formal object oriented design model. In this case, Eiffel provides a subclassing mechanism, but does not fully support the subclassing requirements.
- C++ claims to offer polymorphism when what it actually offers is the dynamic binding of code to message requests. It does not offer replacement polymorphism.

These types of subtle semantic differences plague the process of translation between languages with different semantics, particularly when the target implementation language semantics are not formally defined.

6.2.3 A Review of OOP Languages

It is not possible to review all available object oriented programming languages. Five of the most popular languages, namely Simula, Smalltalk, C++, Eiffel and CLOS, are considered in sections 6.2.3.1 and 6.2.3.2, below.

6.2.3.1 Overview of Language History

This section gives an introduction to each of the five languages by giving a brief review of their histories.

- **Simula** was designed in 1967 as an extension to Algol 60 [91]. It is a general purpose language which, although often ignored by object oriented programmers, is still widely used. Simulation is just one application of Simula.
- **C++**, an extension of C, was designed by Stroustrup in 1984 [106]. It is widely distributed in many forms (by commercial vendors and as public domain software). It is likely to be

the dominant object oriented programming language of the 1990s. The main weaknesses of C++ are: the lack of substitution polymorphism, its hybrid nature and its lack of support for organising libraries (different library classes often turn out to be incompatible). As with C, the syntax is awkward and difficult to parse. A good reference to the latest versions of the language is [2]. C++ is still evolving but there are standardisation efforts. Unfortunately, at the present, the semantics of C++ are such that formality is out of the question.

- **Smalltalk-80** was the first popular object oriented language, developed at Xerox Parc by Kay, Goldberg and Ingalls [58, 57]. It is best known for its contribution to the development of graphical user interfaces, and the manner in which it provides a programming environment (set of complementary tools) rather than just a programming language. Its purity is taken to extremes: all things are objects (even classes). This consistency, paradoxically, can be quite confusing to beginners and experts alike. It is an interpreted language which does not perform any strong type checking. It is a good language for learning about object oriented programming, but it is not suitable for large scale software development.
- **Eiffel** was developed by Bertrand Meyer in 1988 [84], in response to the need for a stongly typed but dynamically bound object oriented programming language. It has many innovative features and appears to provide all that one would require in an object oriented language, but it has its problems (particularly in its implementation). Eiffel is examined in more detail in section 6.4, as the language chosen for implementing object oriented LOTOS designs.
- **CLOS** (Common Lisp Object System) is an extension of common Lisp (see [73, 42], for example). It was developed to include the best features of a wide range of Lisp-based object oriented languages (e.g. Flavours [88] and CommonLoops [7]). Although it is a hybrid language, the object oriented language constructs are so well integrated with the Lisp features that it can be treated as if it was pure. CLOS adheres to the Lisp philosophy of flexibility: it is weakly typed and encapsulation is not enforced.

6.2.3.2 Comparing Characteristics

The table in figure 6.2 identifies the three types of characteristic (esssential, important and beneficial) each of the five languages support and, in appropriate cases, the degree to which they are supported.

6.2.4 Choosing Eiffel

Examination of the table in figure 6.2 clarifies the reasoning behind choosing Eiffel for implementation of the formal object oriented LOTOS designs: it is the only language, under consideration, which fulfils all the essential requirements[6]. Eiffel is not ideally suited to our needs (see section 6.4), but is the *best* option available within the timescale of the thesis. Currently, work is being done towards automating the generation of Eiffel code from object oriented LOTOS designs. This thesis reports only on the manual production of code.

[6] C++ was originally used in the case study, together with Eiffel, but there were great problems with its lack of substitution polymorphism and its informal, yet very complex, definition.

language / characteristic	Simula	C++	Smalltalk-80	Eiffel	CLOS
Classification	Direct	Direct	Direct	Direct	Direct
Encapsulation	Direct	Direct	Unsupported	Direct	Unsupported
Composition	Direct	Direct	Direct	Direct	Direct
Subclassing	Single	Multiple	Single	Multiple	Multiple
Polymorphism	Direct	Unsupported	Direct (ad hoc)	Direct	Direct (ad hoc)
Strong Typing	Direct	Direct	Unsupported	Direct	Unsupported
Compilation	Yes	Yes	No	Yes (incremental)	Yes
Genericity	Unsupported	Direct	Not Applicable	Direct	Not Applicable
Library Support	Minimal	Excellent	Excellent	Excellent	Minimal
Assertions	Unsupported	Supported	Unsupported	Direct	Supported
Garbage Collection	Yes	No	Yes	Yes	Yes
Industrial Acceptance	Minimal	Yes	Little	Little	Minimal
Packaging	No	Limited	No	No	Yes
Concurrency	Direct	Supported	Supported	Supported	Unsupported
Tool Support	Average	Good	Good	Good	Average
Persistency	Unsupported	Supported	Supported	Direct	Unsupported
Purity	Hybrid	Hybrid	Pure	Pure	Hybrid(used purely)

Figure 6.2: Characterising Object Oriented Programming Languages

6.3 Translating Design To Implementation: Mapping Semantics

6.3.1 Implementation Languages: The Importance of Semantics

Good programmers understand the semantics of their chosen implementation language. Bad programmers suffer from a lack of semantic understanding: they must continually check their understanding of the language.

6.3.1.1 The (Reluctant) Acceptance Of Informality

The high-level object oriented designs are constructive in nature and, as such, have potential for direct compilation. A direct compilation approach is not advocated in this work because it is necessary for programmers to be able to manipulate and interact with the implementation code, and the object oriented LOTOS code (in its present form) is certainly not suitable for use by object oriented programmers. Consequently, this thesis advocates using a different language for implementation. There are difficulties which arise from this approach:

- The chosen implementation language Eiffel, like most object oriented programming languages, has no formal semantics. Consequently, there is no way to prove that the code is a valid implementation of the design (and therefore fulfils the initial requirements). Certainly, having

a formal object oriented design helps to test the code, but it cannot guarantee correctness.

- The informal semantics of Eiffel appear, at first glance, to correspond to the semantics of the object oriented LOTOS designs (using the procedural communication model). However, there are many differences between the design semantics and implementation language semantics (these are covered in more detail in section 6.4). The naive view is that the design and implementation languages share a common semantics (i.e. object orientedness) and this can lead to many problems. Implementations may appear to fulfil their requirements (as specified in the design), but without formal semantics these appearances can deceive.

These problems arise from informality in the programming languages. Programmers can, in principle, do what they want with the design provided they can verify their executable model against the requirements. In practice, this is of course impossible. A formal design phase is needed in software development to ensure the requirements are correctly stated in the design. Formally defined programming languages are necessary to guarantee the correct implementation of the design.

6.3.1.2 Object Oriented LOTOS: A Formal Executable Model For The Future?

An alternative to mapping the LOTOS designs to an existing object oriented programming language is to create a new formally specified programming language (based on the object oriented design semantics). Such a language could be a simple syntactic sugaring of the LOTOS object oriented design style of specification. Using a formal implementation language retains formality in the step from design to implementation. However, it was not the approach taken in this thesis:

- The thesis argues that formal object oriented development of software systems using LOTOS is possible. It is easier to show this by mapping the designs towards well accepted object oriented programming environments than by producing such an environment from scratch.
- It is hoped to transfer this work to industry. Industrial acceptance is difficult to achieve without a well accepted base: in this case, a mature programming language and environment provide a foundation upon which industrial interest can be developed.
- By carrying out the implementations in Eiffel, it is possible to show that our approach to software development is practical, whilst also emphasising the problems which can arise when the implementation language has no formal semantics.

There is a need for a formally specified object oriented programming language. A natural extension to this thesis is the development of a programming language based on the object oriented semantics of the formal designs.

6.3.1.3 The Remote Procedure Call Communication Model: An Easy Target

Wegner defines four fundamental types of object: functional, server, autonomous and slot-based, according to their external and internal communication models [124]. Eiffel objects are of the server type: the objects are active only when a message is received that triggers the object's internal

operations. These internal operations are themselves sequences of service requests to state attribute objects of the server. Consequently, there is a single thread of control in an Eiffel implementation. Control goes from client to server and returns to the client after the server updates its internal state and/or returns some result.

6.3.2 Peculiarities of LOTOS Designs

There are many aspects of the LOTOS designs which are peculiar to the approach advocated in this thesis. These must be kept firmly in mind when deriving object oriented code.

6.3.2.1 Sharing

In LOTOS designs whose communication models are procedural, there is no notion of shared objects. For example, when an object `A` is composed from components `B, C` and `D` then there is no access to `B, C` and `D` except through `A`. In effect, all external events of `B, C` and `D` must synchronise with the control process of `A`. This is illustrated in the left hand side of figure 6.3. Contrastingly, in object oriented programming languages, it is common to be able to define components as references to objects[7]. This is illustrated by the right hand side of figure 6.3.

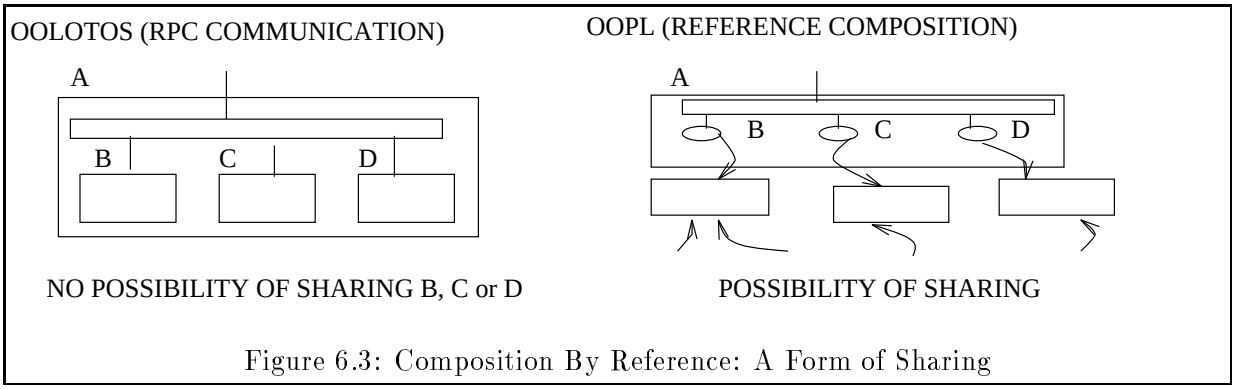


Figure 6.3: Composition By Reference: A Form of Sharing

Consider a LOTOS design of a database enquiry system. There are two components: the database and the interface (which interprets user interactions). A new multi-user system is required to allow parallel access to three users, for example. This type of behaviour is most naturally implemented using sharing, even though it cannot be specified in that way using the formal procedural communication model. Rather than having all access to the database to be through the system control, sharing permits each individual interface to have a reference to the database (and thus have direct access to information). The shared database system can be said to be composed from a number of single-user database components. The underlying components in the design and implementation are the same: the three interfaces and one database. However, the way in which they communicate and interact is very different. This is illustrated in figure 6.4.

Sharing is a powerful implementation mechanism, but it breaks the principle of encapsulation. The state of each of the components of the shared database implementation can be accessed without

[7] In Eiffel, this is the only way of defining object components.

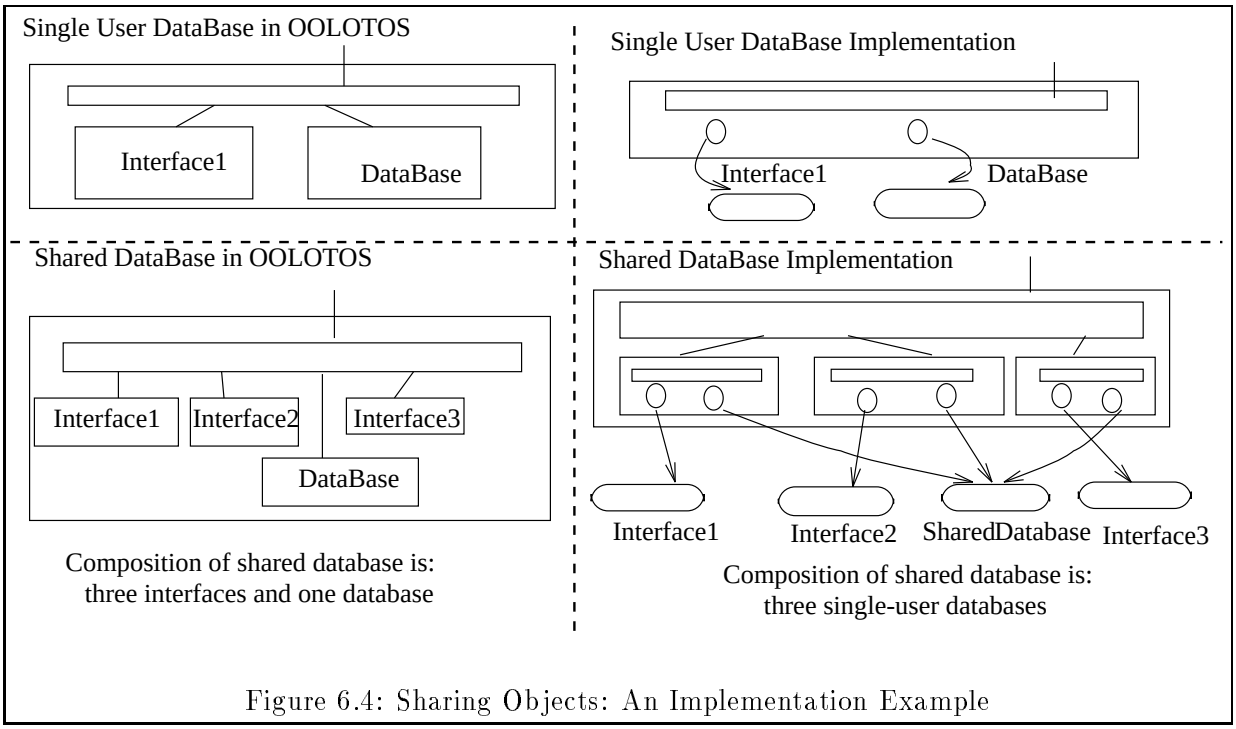


Figure 6.4: Sharing Objects: An Implementation Example

using the component's interface. Alternatively, one could argue that the database is not actually a component of any of the components but is a component of the whole multi-user system (a global variable). Sharing is examined in more detail in section 6.4, when the Eiffel reference semantics are considered.

6.3.2.2 Polymorphism: Parameter Replacement

The OO ACT ONE requirements model states that all parameters of an operation can be actualised by an instance of the specified class or by an instance of a subclass of the specified class. This property of object oriented systems is known as polymorphic replacement. The ACT ONE model specifies polymorphic replacement using coercion and operation overloading. The transfer of ACT ONE structure to the formal process algebra is complicated by the polymorphic requirements[8].

Polymorphic replacement must be considered during implementation. In some languages, this type of polymorphism will be provided for automatically. However, in languages whose semantics do not provide this polymorphic property, it is necessary to code it explicitly where necessary.

6.3.2.3 Implementation Freedom

In the analysis model, a service request is defined as the evaluation of a **state label expression**. When this is translated to ACT ONE, a service request is defined as the evaluation (simplification using standard re-write rules) of an ACT ONE expression. Now, when a system services a request by

[8]The main weakness of using LOTOS for our object oriented design language is that it does not directly support replacement polymorphism.

using its components, these expressions will be constructed from subexpressions which correspond to internal service requests. Often, the order in which these subexpressions is evaluated is arbitrary. This is specified using the interleaving operator (|||). In a procedural implementation, the implementer must resolve this freedom of implementation feature by removing the internal parallelism. This does not change the external behaviour.

6.3.2.4 Exceptions

Exceptions, in the LOTOS designs, are handled in a very distinctive manner: exceptions are returned (as **unspecified** results) to the service requesters (clients) rather than resulting in a run-time error. For example, just because a stack is empty should not prevent it from servicing pop requests, otherwise static analysis cannot, in general, guarantee the absence of a run-time error due to an empty stack receiving a pop request. In the stack case, the designers can either chose to explicitly handle the exception or they can leave the programmers to cope with it. Exceptions are not necessarily error cases, they represent some abstract behaviour which is to be made concrete at less abstract stages of development. Implementers must handle all exceptions in a consistent and coherent fashion. Having exceptions in the design is useful if the target implementation language provides a mechanism for handling them.

6.4 Producing Eiffel from Procedural Object Oriented LOTOS Designs

6.4.1 Setting Reasonable Bounds

Implementing object oriented requirements using Eiffel is a large area of research in its own right. Further, the production of executable code for any given formal specification is a non-trivial task. It is not possible within the implementation part of this thesis to examine formal code generation in any detail. Rather, we set reasonable bounds for the implementation.

6.4.1.1 Restricting Designs to The RPC Communication Model

Rather than attempting to show how Eiffel can be used to implement any given LOTOS design, we restrict ourselves to those written using the procedural communication model.

6.4.1.2 Restrictions on the Eiffel Syntax

Eiffel is a large, complex language with many mechanisms, not all of which are a direct consequence of an object oriented philosophy. Rather than attempting to examine the complete language, only those aspects which are directly relevant to the implementation of formal designs are considered. This simplifies the process of code generation, but means that the code produced in this way may not be the most efficient. It is beyond the scope of this thesis to consider all the ways programmers can tune their code whilst retaining correctness, particularly with respect to the choice of certain language

constructs over others. We are primarily concerned with producing code which fulfils its functional requirements.

6.4.1.3 Emphasis On Semantics

This section does not attempt to analyse the suitability of Eiffel as an object oriented programming language. Such an analysis necessitates:

- A study of Meyer's object oriented philosophy.
- A critique of the Eiffel environment (the language implementation and tools) with respect to their practical application in large scale software development.
- The undertaking of a variety of case studies using Eiffel.

Meyer, of course, gives his opinions on these aspects. A more objective view is given in [121].

6.4.1.4 The Language Version

Eiffel is continually being updated and errors corrected. Meyer has listened to much of the criticism of the language and attempted to make improvements. Unfortunately, it is not possible to always have the most up-to-date version of the language. Further, it is often best to stick to using an older version, rather than continually changing ones understanding of the semantics. Many aspects of the language have remained constant, whilst other important features of the language are very unstable. Analysis of these features is restricted to an early version of the language, namely version 2.3. The reason for this being that the coding in the case study (see chapter 7) was carried out over a year before this research was written up. The newest version of Eiffel is defined by Meyer in [86].

6.4.2 Coding Design Requirements in Eiffel: An Overview

The production of Eiffel code is considered in three main sections:

- Modelling object based requirements.
- Modelling object oriented requirements.
- Utilising assertions and exceptions.

Then, some other interesting aspects of Eiffel are considered. This work is usefully preceded by a comparison between the semantics of Eiffel and the semantics of the object oriented LOTOS designs.

6.4.3 Reference Semantics vs Value Semantics

The object oriented LOTOS and Eiffel have fundamentally quite different semantics, even though they both have an object oriented flavour. The mapping between the two languages appears quite straightforward until the semantics are studied in more detail. Call-by-reference semantics, as is prominent in Eiffel, is appropriate for the specification of an executable language in which *how* not *what* is a prime concern. Further, call-by-reference gives much more control to the programmer with

respect to efficient allocation and use of resources. The value semantics, as is evident in the formal designs, is appropriate for the specification of behaviour at a higher level of abstraction (and whose efficient execution is not a prime concern). Before addressing the problems of relating the two different semantics, it is useful to examine the relative merits of each.

6.4.3.1 Advantages of Reference Semantics

The advantages of reference semantics arise from the extra control given to a programmer with respect to the way in which memory is utilised. By using references, programmers can explicitly access and manipulate state rather than state variables. This is a powerful facility which is often abused. Eiffel reference semantics supports two very powerful programming techniques:

- **Sharing**
  Sharing is necessary when state attributes of different objects must refer to the same object: as opposed to distinct but identical objects. Sharing leads to an economy of space, efficient memory access and update, and semantic integrity (if something in the shared object changes, then this change is simultaneously reflected in all the clients of the shared object). To implement sharing, state attributes are declared as references to other objects. In Eiffel, all state attributes (other than those of simple types) are implemented as references. The global scope of references in Eiffel means that any object can be referenced by any other object. A limited form of sharing control is provided by the Eiffel constant references and the **once** construct. This allows objects to be shared amongst instances of a particular class (and no other).
- **Linked data structures**
  All programmers are familiar with the notion of linked data structures: stacks, lists, trees, etc .... Linked data structures are most useful when there is recursion, or even self reference. They provide the most efficient means of constructing large data stores, with high degrees of control over how the structure is traversed. Linked data structures offer a natural way of implementing recursively defined class structures.

6.4.3.2 Disadvantages of Reference Semantics

Many complications arise when using references:

- **Creation and Initialisation**
  Because the state attributes of an object can themselves be references to other objects, object creation and initialisation must be done in two steps: a declaration (for example, `x:X` declares `x` to be a reference to class `X` and sets the state of `x` to be **void**) and an association (for example, `x.Create` creates an object of class `X` and associates it with reference `x`). There is a confusing duality between the references and the instances.
- **Memory Management**
  Object instances may, at run time, be unreferenced. It is necessary for this state to be made available for re-use (garbage collection). Eiffel provides an automatic garbage collection facility

(as opposed to requiring the programmer to handle it), but it does make the code produced much less efficient.

- **Dynamic Aliasing**
  It is dangerous to have one object which is accessed through two different references. In particular, when references are passed as operation arguments (external attribute parameters) it is not possible to guarantee that the execution of the routine does not change the state of objects other than the one currently servicing the request.
- **Testing for equality and copying**
  With references there are two ways of defining equality: by reference, or by 'state'. Testing for equality of 'state' can be shallow (where all fields are tested for equality of reference) or deep (where all fields are tested for equality of 'state'). In some complex, recursive data structures it is necessary to define even more complicated equalities. The same complexities arise when one considers defining assignment and cloning.

6.4.3.3 References and Values: a Logical Equivalence

It is very simple to implement call-by-value semantics in a call-by-reference language: all calls to a reference are simply replaced by making a deep copy of the object being referenced and passing a reference to the new copied object. In this way, the absence of sharing is guaranteed.

It is also simple to implement call-by-reference semantics using a call-by-value language: every object is uniquely identified and kept (together with its identification) in some global data store. The object identifiers can then be used as state attributes of other objects. Access through identifier can be provided by a global system function which is visible to all objects.

The advantage of a reference semantics is that the global state (and global means of allocation and access) are provided by the language rather than needing explicit control by the programmer.

Reference semantics are very powerful, but do make the production of correct code much more difficult. When implementing the formal designs we do not advocate the 'do everything by value' approach. However, we do not wish every object to be made available for sharing. Consequently, we advocate the use of sharing only in special cases and code the Eiffel so that, by default, operation arguments are passed as references to copies rather than sent as references to the actual parameter. In special cases, sharing can be contained within predefined classes: trees, lists, rings etc .... For example, a linked list structure can be used to implement the following `List` class behaviour.

```
Class List USING listelm OPNS
LITERALS: empty
STRUCTURES: ListStr <List, listelm>
...endclass (* List *)
```

An Eiffel implementation has a structure as defined below.
The object, `AList = List(List(List(empty,3),2),1)` can be created in Eiffel as follows:

```
AList, BList, CList, DList:  List
CList.ListCreate(DList, 3); -- By default DList is empty
```

```
class List ...
feature
Next:  List;
Elm:  ListElm; ...
end -- class List
```

```
BList.ListCreate(CList, 2);
AList.ListCreate(BList, 1);
```

6.4.4 Coding Object Based Requirements

6.4.4.1 A Class Instance Is A System

Meyer states that:

> 'The absence of a notion of main program and of any structuring mechanism at a higher level than the class is an important element of the Eiffel software design philosophy.'

Each class is an executable entity in its own right. The process of creating a system from a class is called assembly. A system is characterised by a *root* class. System execution is done in two steps: *root* declaration and execution of the *root* class *Create* routine.

This suggests that every class can somehow be instantiated and executed. However, in most cases, the creation routine of a class just instantiates the state attributes. Such a creation routine shall be known as a **base creation**. The execution of a **base creation** will not produce a system that performs any useful purpose (other than its very existence). Classes which are intended to be systemised (i.e. turned into systems) are more commonly defined to have a creation routine which acts as a type of main program.

It is not desirable for all classes to have create routines that act as programs in their own right. It is desirable, however, that system classes can be generated from any given class corresponding to a component in the formal design. Object oriented Eiffel implementations generally have a complex root creation routine which provides the user-interface to the system. The coding of such an interface is important for the system, but it is not important for subcomponents of the system. Rather than having programmers define system containers for every class, it is useful to provide a default mechanism to produce a primitive system with minimal human-computer-interface (HCI) features. Such a mechanism is very easy to develop.

6.4.4.2 Defining Class Members

Class members are defined by the different states which objects of a class can attain. In other words, objects are references to particular class members, and every object of a class is uniquely identified with one class member. In Eiffel, every class has a fixed set of state attributes. This is inflexible:

- Literal values cannot be directly defined.
- Classes with 2, or more, structures cannot be directly represented.

Literal values are the object oriented equivalent of enumerated types. Eiffel does not support enumerated types: they must be declared as literal constants (normally integers). For example, the class `t-light`, with three literals `red, amber` and `green`, is implemented in Eiffel below.

```
class t-light ...
feature
red:  INTEGER is 0;
amber:  INTEGER is 1;
green:  INTEGER is 2;
t-light-state:  INTEGER ...
```

Multiple structures can be implemented by defining a structure enumeration to identify the structure (or literal) currently being used as the state constructor of the object. This requires the parameters of every structure operation to be included in the state attribute set (of the Eiffel code), with only a subset of the parameters having meaning at any particular moment in an object's life-time. For example, consider the partial behaviour of the `Student` class, defined below.

```
CLASS Student USING ...OPNS
LITERALS: unregistered
STRUCTURES: single<subject>, joint<subject, subject> ...
```

This is implemented in the Eiffel `Student` class below.

```
Class Student ...
feature
unregistered:  INTEGER is 0;
single:  INTEGER is 1;
joint:  INTEGER is 2;
student-state:  INTEGER;
single-subject1, joint-subject1, joint-subject2:  subject; ...
```

The state of an object is determined by the `student-state` attribute, together with the relevant parameter attributes. The naming convention for the structure parameter state attributes makes it simple to identify the role of each state attribute: the structure name is followed by the parameter class name and index. (Note that there is a redundancy when different structures have common parameters, but it is better to keep the redundancy as it improves the mapping between specification and implementation.)

6.4.4.3 Defining Class Interfaces: Exporting Eiffel Features

Eiffel groups state attributes and operations (methods) together as **feature**s. By default, all **feature**s are private (not part of the external interface). Public features must be listed in the **export** clause at the beginning of the class definition. Meyer does not distinguish between state attributes and operations because, as he correctly argues, it does not matter whether a feature is stored as a state or provided as an operation. This is certainly true, but we believe that distinguishing between state and

operation is important since the state structure corresponds to our notion of composition. The state attributes can still be offered as features by defining particular operations for this purpose. This is the approach taken in our Eiffel implementations: none of the state attributes are exported (directly) and there is a one-to-one correspondence between the exported Eiffel features and the external attributes of the design classes. Eiffel procedures are used to implement transformer attributes, whilst Eiffel functions are used to implement duals and accessors: duals are implemented as functions with side-effects.

A simple stack class illustrates the way in which an object oriented class specification is implemented in Eiffel. The OO ACT ONE code for the `Stack` class is given below.

```
Class Stack USING Int OPNS
LITERALS: empty
STRUCTURES: St<Stack, Int >
TRANSFORMERS: push<Int>
DUALS: pop -> Int
EQNS
Stack1.push(Int1) = St(Stack1, Int1)
empty.pop = empty AND 0; (* ignore exceptions, for the moment *)
St(Stack1, Int1).pop := Stack1 AND Int1
ENDCLASS (* Stack *)
```

This behaviour is implemented in Eiffel below.

This simple implementation of stack behaviour illustrates some interesting points:

- The `INTEGER` base type is used to provide the behaviour of class `Int`. Base types are reconsidered in section 6.4.7.
- The `Create` operation is the default means of initialising the state of an object. Eiffel does not permit creation routines to be overloaded and so we cannot define one creation routine for each literal and structure of a class. In some specifications, like the one above, it is useful to be able to create an object as an instance of a literal or structure. In such cases, we define operations *LiteralName*`Create` and *StructureName*`Create(...)` in an appropriate fashion.
- The code for the implementation of each of the external attributes (`push` and `pop`) is more complex than the specification code because extra care is needed when coding with references in Eiffel.

6.4.4.4 Composition

Every structured object in the formal design is represented as a process in which the structure is either specified in the ADT part, or specified as a set of component processes. The structured objects in the formal designs can be said to contain their components. In Eiffel the components (state attributes) are defined as object references. Therefore, to implement containment (encapsulation) we implement every state attribute as a unique reference which is never passed out of the containing object. In this way, complex Eiffel systems (without sharing) are given a conceptual composition hierarchy based on encapsulation. This is illustrated in figure 6.5.

```
class Stack export push, pop
feature
empty:  INTEGER is 0;
St:  INTEGER is 1;
Stack-State:  INTEGER;
St-Stack1:  Stack;
St-Int1:  INTEGER;
emptyCreate is do Stack-State := empty end
StCreate(Stack1:  Stack, Int1:  INTEGER) is do
      Stack-State := St;
      St-Stack1 := Stack1;
      St-Int1 := Int1 end
pop is local tempstack :Stack do
if Stack-State = empty then Result:= 0 else
      tempstack.Create;
      tempstack:= Current;
      St-Stack1:= tempstack.St-Stack1.St-Stack1;
      St-Int1:= tempstack.St-Stack1.St-Int1;
      Stack-State:= tempstack.St-Stack1.Stack-State;
      Result:= tempstack.St-Int1
end; -- pop
push (int1:  INTEGER) is local tempstack:  Stack do
      tempstack.Create;
      tempstack := Current;
      Stack-State:= St; St-Stack1:= tempstack; St-Int1:= int1;
end; -- push
end -- class Stack
```

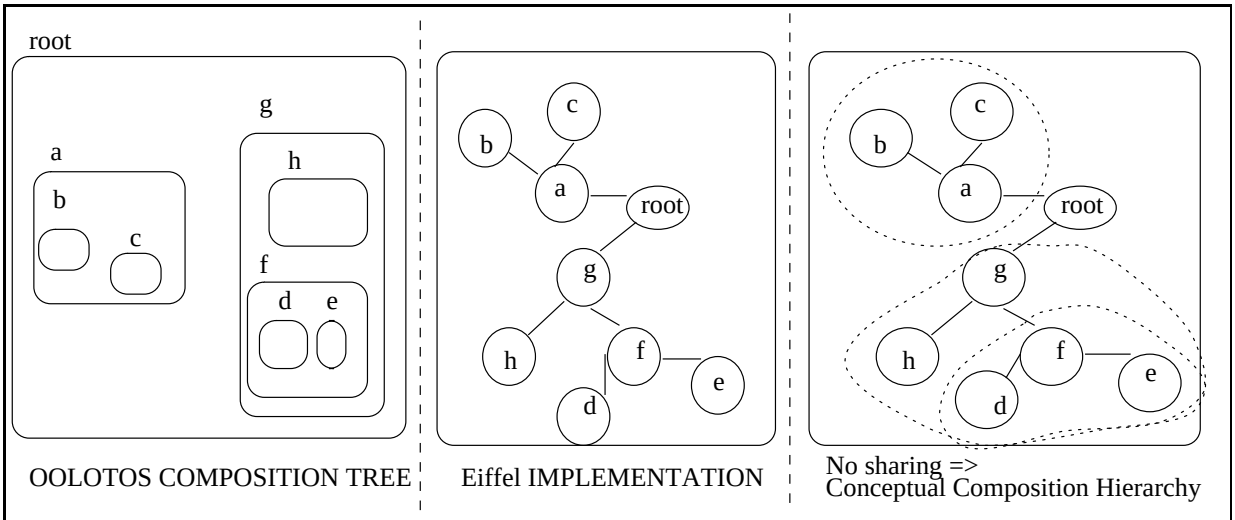


Figure 6.5: Composition in Eiffel

By convention, unshared component attributes are commented as: `-- components`. The pro-

gramer must then ensure that such references are never passed outside the containing class. In the special cases where sharing is required, or a linked data structure is being used, then these shared attributes must be commented with an explicit statement of why and how the reference is being shared (this is good programming practice).

6.4.4.5 Implementing State Changes

The state changes of a structured process are implemented as follows:

- **Restructuring**
  This is implemented in Eiffel by changing the enumerated structure value and updating the other state attributes appropriately.
- **Pure Updates**
  These occur when the structure representation remains the same and the state of the system changes only as a result of service requests to component objects changing the components' state. In Eiffel, this means that the state attribute references do not need to be directly manipulated.
- **Impure Updates**
  These occur when the structure remains the same, but at least one of the components changes because of direct manipulation. In Eiffel this can only be implemented by an explicit change to one or more of the state attribute references.

During design, emphasis is placed on purity: such classes can be very simply implemented in a reference semantics because no direct manipulation of the referenced component attributes is necessary. Impure state changes are prone to errors because the state attribute references need re-allocation. In some cases, the extra work required of programmers does not justify the flexibility of allowing impure changes to state. Contrastingly, restructuring of state corresponds to special events in the lifetime of an object and therefore the additional work required on the programmers part is justified.

6.4.4.6 Implementing Encapsulation

It seems strange, when encapsulation is fundamental to object oriented programming, that object oriented programming languages use reference semantics in such an uncontrolled fashion. The Eiffel implementations of the formal designs use shared references only in very particular instances. Sharing cannot be discarded, but should be used only with due care and attention. Implementations which used shared references must always contain the sharing within well understood encapsulated behaviours. When such behaviours are re-used there is no evidence of the sharing at the component interface.

6.4.5 Coding Object Oriented Properties

Subclassing (extension and specialisation) in the formal object oriented designs is concerned with two aspects of behaviour:

- Behaviour Compatibility: An instance of a class always offers equivalent behaviour as the corresponding instance of the superclass.
- Polymorphic Replacement: A subclass is type compatible with all its superclasses, i.e. type checking involves checking the class of actual parameters against all the subclasses of the formal parameters (including, of course, the class of the formal parameter).

Inheritance, in object oriented programming languages, is primarily concerned with code re-use and dynamic binding. Both these are efficiency concerns: the faster production, compilation and execution of code. However, the ability to override inherited methods means that behaviour compatiblity is no longer guaranteed.

6.4.5.1 Multiple Inheritance

Both OO ACT ONE (and the LOTOS designs in which OO ACT ONE is used) and Eiffel offer multiple inheritance. However, multiple inheritance in Eiffel has many associated difficulties which are not present in the design language. These must be addressed when implementing the formal requirements:

- **Repeated Inheritance**
  All multiple inheritance systems must be able to cope with repeated inheritance: when a class has a superclass which can be reached by more than one route up the class hierarchy. Eiffel resolves this problem by adhering to the following rule:

  > *In repeated inheritance, any feature from a common parent is considered shared if it has not been renamed along any of its inheritance paths. Any feature which is renamed at least once is considered replicated.*

  Consequently, when implementing repeated inheritance properties it is necessary for the Eiffel programmer to rename some features.
- **Name Clashes**
  Languages with multiple inheritance must deal with name clashes, i.e. when features inherited from different superclasses have the same name. In Eiffel, name clashes are forbidden. This places the onus on the programmer to cope with name clashes (using a construct for renaming inherited features) and weakens ones ability to re-use classes (by inheritance) without undue problems. This renaming approach is definitely not an ideal solution [121] and newer versions of Eiffel [86] allow explicit routing qualification to cope with name clashes.

6.4.5.2 Implementing Extension and Specialisation

Implementing the extension and specialisation relationships in Eiffel appears to be straightforward:

- **Extension**
  An extension class is defined by including the new function in the subclass, and perhaps redefining other features to take advantage of the new function for improved efficiency.

- **Specialisation**
  A specialisation is declared by defining a new invariant of the superclass. This invariant identifies the particular state partition of the superclass which makes up the new state of the subclass. The subclass state enumeration is useful in this respect.

This subclassing seems to be a perfect match until one considers the rules of contravariance and covariance. The formal design semantics state that:

- The class parameters of a structure operation of a subclass may be defined as subclasses of the corresponding parameters in the superclass.
- The result of a valued transition in a subclass may be defined as a subclass of corresponding result in the superclass.
- The parameters of a transition in a subclass may be defined as superclasses of corresponding parameters in the superclass.

Unfortunately, Eiffel does not support the last contravariance property. Cardelli [22] provides an in-depth study of the reasons for and against contravariance. Cardelli advises that subclassing must support contravariance otherwise the polymorphic replacement property is not guaranteed to maintain correctness. Unfortunately, contravariance is very difficult to implement cleanly in Eiffel: it is necessary to override the Eiffel typing system with ones own (as in an imperative implementation) and graft this onto the Eiffel class hierarchy (a non-trivial task). Consequently, we chose to ignore the need for contravariance and keep it in mind for future development.

6.4.5.3 Typing Problems

The type checking of the formal designs should, in theory, guarantee type checking correctness of the resulting Eiffel (except the instances of contravariance in subclassing definitions). However, implementation often produces code which requires its own type checking. Eiffel type checking is very complex. The implementation of Eiffel, to date, cannot cope with the checks required by the informal semantics. This is design oversight. Cook [30] analyses the holes in Eiffel type checking and proproses removing some Eiffel flexibility as a solution. The thesis by Dinesh [47] analyses Eiffel type checking with respect to incremental development.

6.4.6 Using Eiffel Assertions and Exceptions

Assertions define properties of some value(s) of program components. The Eiffel assertion language is not as powerful as full predicate calculus but it does provide a means of defining simple boolean expressions.

Assertions take three forms in Eiffel:

- **Preconditions**
  The **require** construct places a precondition on a routine being executed. This is not important when implementing the formal designs because the class instances are always able to service all requests in their external interfaces.

- **PostConditions**
  The **ensure** construct is used to guarantee some property after a request has been serviced. Postconditions can be used to record some of the abstract behaviour defined for the class in which the routine is found. For example, after a `push` on a `Stack`, a suitable postcondition is the boolean expression `Stack.notempty`.
- **Class Invariants**
  Preconditions and postconditions describe only properties associated with individual routines. We require, when implementing formal requirements, a means of implementing class invariants: global properties of class instances. Eiffel provides a **class invariant** construct. Class invariants are useful in our implementations because they are checked on creation and at all stable times in the lifetime of an object. Further, they are passed down the inheritance hierarchy. This is important for the correct implementation of specialisation relationships.

Assertions are advantageous in our object oriented implementations because they:

- Improve the relationship between specification and implementation.
- Act as a documentation aid.
- Can be used to explicitly handle exceptions in a coherent fashion.

In principle, runtime checking of assertions should not be necessary. However, it is currently beyond the state-of-the-art in software development tools to perform a static analysis of such correctness. Consequently, it is necessary, whilst testing, to monitor for exceptions in the Eiffel code at run time. After testing, this monitoring can be turned off to improve performance.

Note that assertions and exceptions are useful for implementers both as a debugging mechanism and as a way of relating requirements with code. However, the Eiffel mechanisms do not constitute a formal approach to software development: the mechanisms are purely syntactic sugar. Eiffel does, however, encourage a methodological approach to exception handling, which can be used to implement the exceptions in the formal designs.

In conclusion, we note the limitations of the assertion language in Eiffel. For example, in the Stack behaviour it is not possible to assert that `pop(push(x,s)) =s`. Meyer acknowledges this problem by stating that this behaviour should be incorporated as a comment. When implementing the formal designs in Eiffel, we recommend that the complete code for each class is included as a comment (or comments) in the Eiffel code.

6.4.7 Other Aspects

Other important issues when coding with Eiffel are:

- **Genericity**
  Inheritance and composition are not powerful enough for general re-use. A mechanism for defining parameterised classes (generic behaviour) is required. Generic classes are simple to define using Eiffel. The mapping from generic ACT ONE types is straightforward.

- **Packaging**
  The lack of a packaging facility is one of the major weaknesses of Eiffel. Much of the abuse of multiple inheritance results from the need to inherit shared features which would be better grouped in some sort of package construct.
- **Simple Types**
  Eiffel simple types (INTEGER, BOOLEAN, CHARACTER and REAL) are called-by-value. They are useful in providing efficient implementations of well-understood behaviour. However, there is one problem with them: they cannot be placed in the class hierarchy. When it is necessary to provide these behaviours in the class hierarchy (for example, one may wish to define a class `EvenInt` as a specialisation subclass of `INTEGER`) then these simple types can be contained within classes which contain the simple types as their only state attributes. This results in a loss of performance, but improves the consistency of the code.
- **Persistence**
  Eiffel supports persistence using classes `STORABLE` and `ENVIRONMENT`. They are very useful, but again inheritance is often abused to utilise these mechanisms. These classes are most useful in the definition of the root system class.
- **External Interfacing**
  The `EXTERNAL` mechanism provides for the incorporation of non-Eiffel code into Eiffel programs. It is beyond the scope of this thesis to examine this mechanism.

6.5 A Question of Concurrency and Distribution

This section is primarily concerned with concurrency, and the potential for application of the formal object oriented development method towards a concurrent implementation. Distribution of implementation resources is also fundamentally a concurrency issue: distributed systems are constructed from concurrent processes[9]. Utilisation of distributed resources can be a performance issue, particularly when the problems being solved are highly concurrent in nature. Parallel code can be used to solve some problems much more efficienctly than others but, in general, this is not the case. There is a much more common reason, other than efficiency, for requiring concurrent software: the external interface of the system being coded may be physically distributed. By considering concurrency in object oriented implementations, the problem of distribution is also being addressed, albeit indirectly.

The preceding sections of this chapter have addressed only sequential implementation, and chapter 5 shows how designs can be targetted towards such an implementation environment. There are three main reasons for concentrating on sequential implementation, at this stage of the work:

- Eiffel does not directly support concurrency[10].

[9]The problem of implementation in a parallel architecture and implementation in a distributed architecture are logically the same, but of course there are additional problems in distributed systems: for example, the inter-process communication is slow and may be prone to errors.

[10]Meyer is currently working towards a *concurrent Eiffel*, but this extension is not available commercially.

- Concurrent semantics are, by their nature, much more complex than procedural (sequential) semantics.
- The current state of the art in concurrent object oriented programming languages are neither robust enough nor well enough supported for the implementation stage of a general software development method.

In general, the following principle should be followed:

> *Don't make object oriented programming more difficult than it needs to be: if concurrency is not necessary then don't introduce it.*

However, if a concurrent implementation is required then this section argues that formal object oriented development using LOTOS can help to deal with it. In particular, the CPT-based design process has great potential for application towards concurrent software development.

Object oriented development adheres to the most elemental engineering principle: *make the solution look like the problem.* Implicit in the formal analysis models is the notion that components of a system are concurrent. Designers must decide whether to ignore or utilise this feature. Generally, designers chose to ignore the concurrency because they know that it is much more complex to deal with than sequentiality. Concurrent designs should be specified only when the target implementation environment is capable of coping with such requirements.

6.5.1 Concurrency and Objects: Opposing Views

The object oriented community is divided on the notion of concurrency. The two extremes to this division can be categorised as *optimistic* (or naive) and *pessimistic* (or conservative).

- **The optimistic view**:
  Object oriented approaches, rather than placing control with some sort of master process, offer a means of handling concurrency which is quite different from traditional approaches. In the object oriented paradigm, objects can 'look after themselves' within a concurrent environment. The shifting of responsibility from centralised control to decentralised control is natural in an object oriented approach. Further, such distributed control can simplify the system.
- **The pessimistic view**:
  Concurrency issues, contrary to initial expectations, are not orthogonal to object oriented concepts. The interference of concurrency and object oriented features makes it difficult, if not impossible, to combine them in a consistent and coherent fashion. In particular, the efficient implementation of concurrent object oriented semantics is in doubt.

The remainder of this section attempts to give a more balanced view of the future of concurrent object oriented languages. In particular, it examines the potential benefits of using formal object oriented development when heading towards a concurrent implementation language.

6.5.2 Concurrency: A Problem of Scale

6.5.2.1 The Problem

Objects may be considered as independent abstract machines that interact in response to service requests. To consider each object as an independent concurrent unit (process) results in an overdose of concurrency: hundreds, thousands or even millions of concurrent entities are required in even the simplest software systems. Implementation environments have not yet reached the stage where such high degrees of concurrency can be adequately dealt with (in hardware or software). Consequently, such a naive approach to modelling concurrency will result in poor performance (if any performance at all). Further, the complex structure of a program is difficult to analyse as a large set of interacting concurrent behaviours.

6.5.2.2 A Solution: Mixing Communication Models

There is a simple way of coping with the problem of scale, without having to reject concurrency as an implementation strategy. Instead of every object in the system being implemented as some sort of independent abstract machine, the designers must explicitly identify (sub)systems of concurrent objects. Then, only particular parts of the system (hopefully the ones whose concurrent implementation would be most beneficial) need be modelled concurrently.

The formal CPT driven object oriented design method provides the flexibility to deal with such a distribution of concurrency. Throughout the LOTOS designs there are, at the moment, three standard internal communication models which structured processes can adhere to: procedural, centralised concurrency and distributed concurrency (see 5.5.5). Object oriented designers can, using LOTOS, mix these communication models throughout the design. Using such a flexible design technique means that concurrent aspects of the requirements model can be explicitly mapped onto concurrent resources in the implementation.

6.5.3 Concurrency and Object Orientation: Resolving Conflicting Requirements

It has been said that concurrency has no respect for the spirit of the object oriented paradigm [95]. Papathomas, in his thesis, identifies five requirements for satisfactory integration of concurrent and object oriented features:

- Mutually exclusive protection of object's state.
- Request Scheduling Transparancy.
- Internal Concurrency.
- Reply Scheduling Transparancy.
- Compositionality and Incremental Modification.

He argues that a concurrent object oriented language cannot offer abstraction, encapsulation and subclassing if it does not fulfil these requirements. However, a concurrent object oriented semantics which fulfils these requirements may not be amenable to efficient execution. Further, such semantics

may not be suitable for use throughout the whole of the development process. We believe that the object oriented LOTOS designs can be used to model object oriented properties and concurrency in a consistent and coherent fashion.

6.5.4 The Future: Formality in Concurrent Compilers?

A design technique in which different communication models are distributed throughout the design is inflexible with respect to matching resources to design components. This matching must be done at compile time and so the mapping of processes to processors is static. Although the area of real concurrent compilers, where the compiler actually maps components of the design onto real hardware resources (chips/processors), is still in its infancy, it is clear that static allocation of resources has major disadvantges:

- When a resource fails, the whole system is affected.
- When more resources become available (which may be the case in a truly open distributed system) it is not possible for the compiled code to take advantage of this.
- As more demand is placed on the set of shared resources, it is not possible to release resources for other users.

Correctness preserving design transformations may hold the key to flexible concurrent compilation:

- Given a static concurrent compiler, it is important that different design models are tested before a final design is chosen for implementation. Such testing can evaluate the performance of the system when the concurrency is distributed in different ways. Then, when the final design is implemented, it is more likely that the resources are used effectively.
- The future for concurrency, and object oriented programming, may be flexible compilation, where the mapping from system component to resource is dynamic. It may be possible to compile the design to produce a virtual machine which can be executed in a number of different forms. The different forms can be virtual designs, composed from virtual processes, whose behaviours are related by a set of correctness preserving transformations. The executable machine must be aware of resource allocation in the implementation environment and change virtual design form in response to an increase in supply (or demand) of implementation resources. Of course, there will be overheads in performing such virtual form changes, but it is possible to envisage a case when such overheads would be negligible compared with the increase in performance. Perhaps some form of process caching, working in the same way as state caching, will become standard in such systems.

The point being made is not that these notions of flexible compilation are new, but that the formal object oriented design framework provides a semantics upon which such compilers could be constructed. Formality is essential in proving that changes of design form (in the virtual machine) do not affect the external behaviour of the system. Correctness preserving transformations appear to be an ideal theoretical tool for reasoning about such models. This (hypothetical) work is beyond the scope of this

thesis, but it acts as a good motivator for using formal methods: such concurrent compilation would be impossible to control without underlying formality. It is difficult to predict the future, particularly with regard to concurrency and objects, but formality is sure to be a play a major role.

Chapter 7

Formal Object Oriented Development: A Case Study

This chapter reports on a case study which investigated the practical application of FOOD. The goal of the case study was to model the requirements of a simple banking network and, using FOOD, to produce an Eiffel implementation of these requirements. The structure of this chapter is as follows:

- **Section 7.1: Introducing the Banking Network Problem**
  This section introduces the case study. The criteria by which the case study was chosen are given. Then, the limitations of the case study are reported. The section concludes by giving an informal overview of the banking network problem, which is the starting point for the development of a formal requirements model.
- **Section 7.2: Formal Object Oriented Analysis of the System**
  This section reviews the process by which a formal OO ACT ONE model of the banking network system is developed. In particular, it illustrates how the analysis and synthesis of a formal model improves mutual understanding between customer and analyst. The opportunistic flavour of the analysis and requirements capture method is emphasised.
- **Section 7.3: Design: Moving the System from the Abstract to the Concrete**
  This section reviews the process by which the banking network requirements were transformed into a high-level LOTOS specification which was ready for implementation in Eiffel. Particular attention is given to the means by which the internal routing of messages was designed for implementation.
- **Section 7.4: The Eiffel Implementation**
  This section records how the Eiffel implementation was developed from the final LOTOS design of the banking network. We emphasise how the analysis and design stages make the implementation process straightforward.
- **Section 7.5: A Review of the Case Study**
  This section examines the lessons which arose out of the banking network case study. Three main aspects of the development process are highlighted: the extendibilty of systems produced

using FOOD, the production of re-usable components at all stages in FOOD and the need to re-evaluate software development planning due to FOOD placing greater emphasis on the early stage of development.

7.1 Introducing the Banking Network Problem

7.1.1 Choosing the Case Study

A banking network was chosen to form the basis of the case study for the following reasons:

- **Familiarity and Understandability**
  The problem of communicating across a network is well understood and there is a wide range of documentation available, for example [109, 102, 39]. Further, the facilities offered by a bank provide a functionality which is accessible to a wide readership.
- **Size and Complexity**
  It is important to chose a case study which is large enough to illustrate the FOOD method, whilst small enough to be effectively presented in this thesis. Although the networking concept is very simple, it does deal with complex issues. Further, banking functionality is neither trivial nor overly complex.
- **Extendibility**
  There has to be scope within the case study for extending and refining the system requirements. The banking network provides two orthogonal dimensions of complexity which can be extended: the architectural and communicational aspects, and the accounting behaviour.
- **Multidimensional**
  It is important that the case study places demands on all three dimensions of software complexity: data structure, data transformation and data communication. Networking is primarily concerned with the communication of data. The banking functionality is complex with respect to the structure and transformation of data. The banking network problem domain provides a case study which naturally combines the complexities of networking and accounting, and so fulfils the multidimensional requirement.

7.1.2 Limitations of the Case Study

The major limitation of the case study is that it does not adequately address the informal aspects of FOOD: in particular, the processes of customer-analyst communication and designer-implementer interaction were not studied. In the case study, the roles of customer, analyst, designer and implementer were played by one person, namely the author[1]. The case study provides a good evaluation

[1] In a preliminary investigation, a network routing model was developed in a project involving both a specifier (the author) and an implementer (David Freer, of British Telecom, must be thanked for his contribution in the development of the C++ code). [55] reviews this preliminary investigation and discusses the process of specifier-implementer communication. It concludes by stating that this communication is improved by the synthesis and analysis of a formal model of requirements.

of the mathematical models, but only a limited evaluation of the methods in FOOD. We believe that a method should evolve from repeated use of models rather than sprouting automatically from the theory. The application of FOOD in one case study does not justify the definition of a prescriptive development method.

The size of the case study was also a limitation: it was not possible to investigate all aspects of the FOOD approach. The case study does, however, illustrate the application of FOOD in a non-trivial problem domain. There is good reason to believe that, given the object oriented nature of FOOD, it can be applied to even larger systems, requiring the co-ordinated attention of groups of developers.

7.1.3 The Scope of the Problem: An Informal Overview of Requirements

The scope of the requirements is represented in figure 7.1. An informal description of the problem domain is given in sections 7.1.3.1 to 7.1.3.4. This informal description forms the basis upon which the analysis and synthesis of the requirements model are initiated.

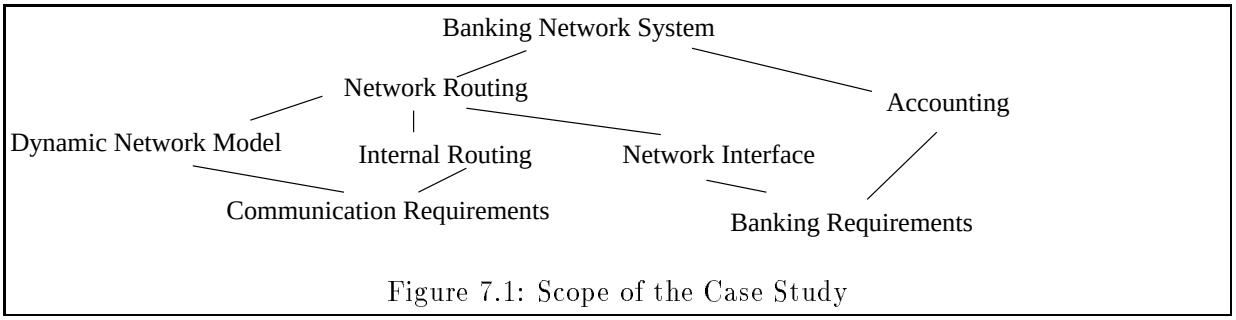


Figure 7.1: Scope of the Case Study

7.1.3.1 Dynamic Network Model

It is a requirement of the system that the network is composed of a set of nodes and a set of links, each of which must connect two of the nodes. The topology of the network must be dynamic: it must be possible to add nodes and links during system execution. The links provide communication lines between nodes along which banking transactions are routed.

7.1.3.2 Network Interface

The interface to the network system is constructed from the interfaces of each of the node components. There are four different types of node, each of which offers its own type of interface:

- **Relay Node**
  A relay node acts purely as a communication buffer for the receiving, routing and sending of messages. Relay nodes offer an interface to system engineers to permit their switching on and off: nodes which are off cannot service requests. All nodes in the system are uniquely identified as relay nodes.

- **Automatic Teller Machine (ATM)**
  An ATM provides an interface to account holders for the reading and writing of account details in response to deposit, withdrawal and enquiry transactions.
- **Control Node**
  A control node provides an interface to bank employees for the purpose of account maintenace. They also act as the access points to a subset of the database of account information. Every account transaction must be routed to one of the control nodes in the network for processing.
- **Teller Node**
  A teller node offers all the functionality of a control node and an ATM. These are used when account holders and bank employees access account information together.

7.1.3.3 Internal Routing of Accounting Information

All account transactions must be routed through a specified control node. We do not require that every transaction is processed, but an *internal* timeout facility must inform the customer (account holder or bank employee) when a transaction has not been processed. The means by which transactions are routed to/from control nodes is a design and implementation decision.

7.1.3.4 Accounting

The database of accounts can be altered by the control nodes in the following ways:

- **Creating a new account**
  A new account is created when the appropriate details are provided, depending on the account type (see below).
- **Closing an existing account**
  An account, specified by a given identification, can be closed only when the balance is zero.
- **Changing the restrictions on an account**
  A **restricted** account (see below) has a limit placed on the size of individual withdrawals. This limit can be changed at a control node.
- **Changing the overdraft facility**
  The overdraft limit for any given account be changed at a control node.

There are three types of account:

- A **basic** account permits the customer to deposit and withdraw money. Further, the customer can request details of the amount of money available to them (i.e. their balance plus their overdraft limit).
- A **business** account permits the customer to deposit and withdraw money, and request a statement of their last three money transactions or the current amount available.
- A **restricted** account permits the customer to deposit and withdraw money. The customer is restricted to withdrawing no more than a predefined amount at any one transaction. A statement of the amount available can also be given for a **restricted** account.

Customers interact with their accounts, which must be uniquely identifiable, from the ATM nodes of the banking network.

7.2 Formal Object Oriented Analysis of the System

The informal requirements, given in the previous section, provide a good overview of the scope of the problem. They also provide the basis upon which a formal requirements model can be constructed. The construction of the formal requirements model plays three main roles:

- It improves customer and analyst understanding of the problem.
- It provides an executable model for customer validation.
- It acts, in its final form, as an input to the design stage.

7.2.1 *What* not *How*

The formal object oriented analysis language, OO ACT ONE, is used to define the accounting functionality offered at the external interface of the banking network system. Further, OO ACT ONE is used to model the dynamic network requirements: *how* these are realised is not an analysis concern. OO ACT ONE is not used to specify *how* the internal routing of transactions, from ATM node to control node, takes place.

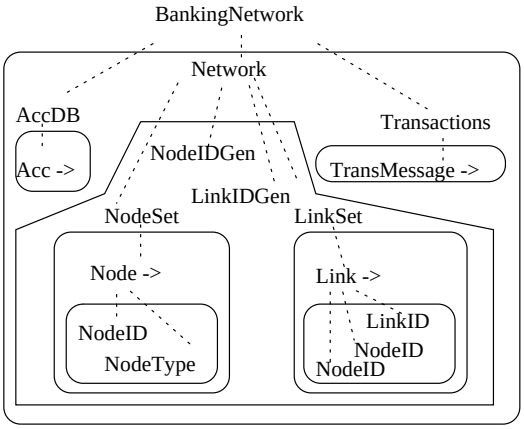
7.2.2 Applying the Skeleton Method to Requirements Capture

Section 4.5.2 defines a *skeleton* method for the synthesis and analysis of requirements models in OO ACT ONE. An important aspect of this method is customer interaction. Although there was no direct customer involvement in the case study, the process of *customer interaction* was replaced by the need for the requirements model to be tested against relevant documentation and intuition. Consequently, in the remainder of this chapter, when referring to *customer interaction* it is this testing process to which we are alluding.

Other than *customer interaction*, all other parts of the *skeleton* method were carried out as intended. The opportunistic aspect of the method meant that there were many ways in which the requirements model could have been constructed and validated. The sequence of steps which was followed is reported below.

Step 1: Composition Analysis of the BankingNetwork System Class

To start, there is only one class to be considered: the `BankingNetwork`. From the informal requirements it is clear that a composition analysis will improve ones understanding of the requirements. Consequently, the analysis is started with a structured decomposition of the problem based on the *has-a* relationship. The diagram below illustrates the initial decomposition of the banking network.

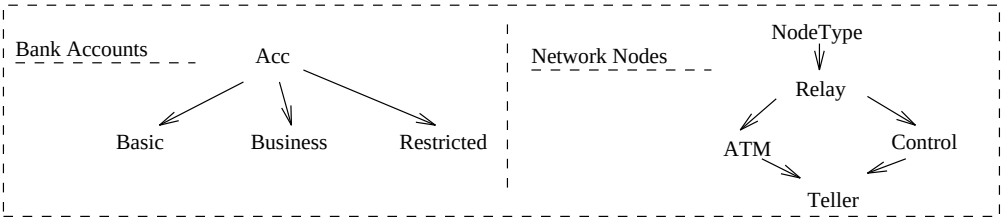


A BankingNetwork is composed from:
An account database (of class AccDB) which is composed from:
Accounts (of class Acc) in a recursive structure
A network (of class Network) which is composed from:
A set of nodes (of class NodeSet) which is composed from:
Nodes (of class Node) in a recursive structure, each Node composed from:
A unique identifier (of class NodeID)
A type identifier (of class NodeType)
A generator of unique node identifications (NodeIDGen)
A set of links (of class LinkSet) which is composed from:
Links (of class Link) in a recursive structure, each Link composed from:
A unique identifier (of class LinkID)
Two node identifiers (of class NodeID)
A generator of unique link identifications (LinkIDGen)
A transaction set (of class Transactions) which is composed from:
Transactions (of class TransMessage) in a recursive structure

Note: A recursive structure, represented as Element ->, is the means in OO ACT ONE of modelling linked lists of Elements.

Step 2: Classification Analysis

Examination of the informal requirements leads to the initial identification of two[2] subclass hierarchies: account classes, specified as class `Acc`, and node classes, specified as class `NodeType`. It is important that these two subclassing relationships are explicitly defined at some point in the requirements capture.



Step 3: Analysis and Synthesis of Network Requirements Model

The network component of the system appears, from the composition analysis, to play a major role in providing the behaviour of the banking network. Consequently, we chose to analyse and synthesise a network requirements model. The behaviour of the network component is defined by an OO ACT ONE `Network` class[3].

Step 3.1: The External Interface of the Network

The composition of the `Network` class has already been analysed in step 1. It is necessary now to consider how the components of `Network` combine to provide network behaviour. An initial OO ACT ONE specification of the `Network` class facilitates further investigation of the requirements. The `Network` header, below, defines the external interface of the class. The header also defines a `Network`

[2] Although there are only two hierarchies identified at this stage, this does not mean that there are only two hierarchies in the system. Step 4.4 identifies another hierarchy which was 'missed' in this early stage. The opportunistic approach to building a formal requirements model encourages the analyst to record understanding even when it is incomplete.

[3] By convention, all class identifiers in the case study have an initial capital letter.

to have a fixed structure[4], i.e. a static set of component classes.

```
CLASS Network USING NodeSet, LinkSet, NodeIDGen, LinkIDGen OPNS
STRUCTURES: ANetwork<NodeSet,LinkSet,NodeIDGen,LinkIDGen> (*FIXED*)
TRANSFORMERS: addNode<NodeType>, addLink<NodeID,NodeID>, switch<NodeID>
ACCESSORS: getNode<NodeID> -> Node, isNode<NodeID> -> Bool,
areConnected<NodeID,NodeID> -> Bool EQNS ...
```

Step 3.2: The Behaviour at the External Interface

The behaviour intended for each of the `Network` attributes[5] is as follows:

- `addNode`: Takes a `NodeType` as an input parameter and allocates it a new identifier (provided by the `NodeIDGen` component). These two values are made into a single `Node`, which is then added to the `NodeSet`.
- `switch`: Takes a `NodeID` as an input parameter and switches the corresponding `Node` in the `NodeSet` from `off` to `on` (or vice versa).
- `getNode`: Takes a `NodeID` as an input parameter and returns the corresponding `Node` in the `Network`, and `false` otherwise.
- `isNode`: Takes a `NodeID` as input parameter and returns `true` if there is a `Node` in the `NodeSet` with `NodeID` as its identifier.
- `areConnected`: Takes two `NodeID`s as input parameters and returns `true` if there is a `Link` in the `LinkSet` which connects the two specified `Nodes`.
- `addLink`: Takes two `NodeID`s as input parameters and adds a new `Link` to the `LinkSet`. The new `Link` is allocated a unique identifier by the `LinkIDGen` component.

When a `NodeID` input parameter does not identify a node in the `NodeSet` of the network then an exception must be defined. (All exceptions must be dealt with during design and implementation.)

A class structure diagram (see figure 7.2) is used to show how the `Network` depends on each of its components to fulfil its external functionality.

[4] By convention, in the case study, all fixed structures of a class `ClassName` are labelled `AClassName`.

[5] The attribute labels have, by convention, a non-capital initial.

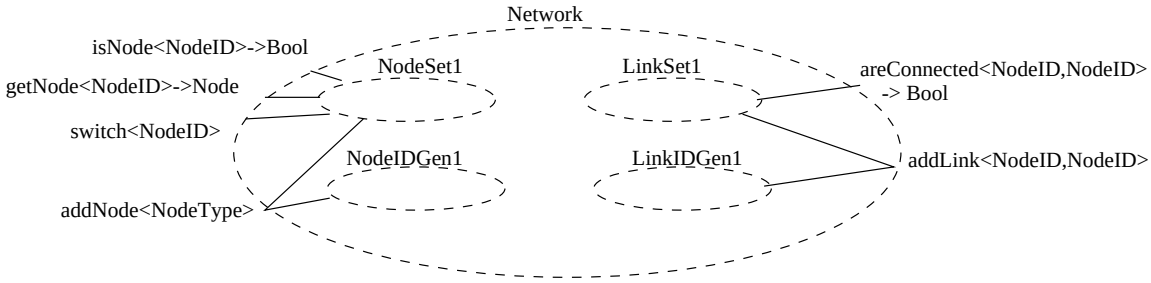


Figure 7.2: The Network Class Structure Diagram

The OO ACT ONE equation definitions for the `Network` class are given below[6].

```
CLASS Network ...(* Header is given above *) EQNS
ANetwork(...).addNode(NodeType1) =
  ANetwork(NodeSet1.add(NodeIDGen1..nextN,NodeType1),LinkSet1,NodeIDGen1.nextN,LinkIDGen1);
((NodeSet1..isNode(NodeID1)).and(NodeSet1..isNode(NodeID2))).not =>
  ANetwork(...).addLink(NodeID1,NodeID2) = ~NodeSet OTHERWISE
  ANetwork(NodeSet1,LinkSet1.link(NodeID1,NodeID2,LinkIDGen1..nextL),
  NodeIDGen1,LinkIDGen1..nextL);
(NodeSet1..contains(NodeID1)).not =>
ANetwork(...).switch(NodeID1) = ~NodeSet OTHERWISE
ANetwork(NodeSet1.switch(NodeID1), LinkSet1, NodeIDGen1, LinkIDGen1);
(NodeSet1..contains(NodeID1)).not =>
ANetwork(...)..getNode(NodeID1) = ~NodeSet OTHERWISE (NodeSet1.getNode(NodeID1))..getNode;
ANetwork(...)..isNode(NodeID1)= NodeSet1..contains(NodeID1);
ANetwork(...)..areConnected(NodeID1, NodeID2) = LinkSet1..areConnected(NodeID1, NodeID2)
ENDCLASS (* Network *)
```

Step 3.3: Analysis of the IDGen Classes

One class, `IDGen`, is defined for the generation of unique identifiers. It is defined to have a `DUAL` attribute, `next`, for the generation of unique identifiers, and an `ACCESSOR` attribute, `eq`, for testing the equality of identifiers. `NodeIDGen` and `LinkIDGen` are defined as renamings of `IDGen`.

`IDGen` must be able to generate an infinite number of identifiers. The simplest way of specifying this is to define an `ID` class with a recursive `STRUCTURE` operation, together with a base `LITERAL` value. Rather than specifying `IDGen` as a store of the previously allocated `ID`s, a standard scheme is employed whereby a unique identifier can always be generated when only the previously allocated identifier is known. Classes `ID` and `IDGen` are defined below. These classes are added to a library for re-use.

`NodeID` is defined, below, using the OO ACT ONE renaming construct.

[6] A simple syntactic sugar is used in the remainder of this chapter to simplify the representation of OO ACT ONE equation definitions in a class with a fixed structure. Rather than writing `Structure(par1,...,parn) = ...` on the left hand side of equation definitions, `Structure(...) = ...` is used without risk of ambiguity.

```
CLASS ID USING Bool OPNS
LITERALS: 0 STRUCTURES: IDSt<ID>
ACCESSORS: eq<ID> -> Bool
EQNS 0..eq(0) = true; 0..eq(IDSt(ID1)) = false; IDSt(ID1)..eq(0) = false;
IDSt(ID1)..eq(IDSt(ID2)) = ID1..eq(ID2)
ENDCLASS (* ----------------- ID ----------------- *)
CLASS IDGen USING ID EXTENDS ID WITH OPNS
DUALS: Next -> ID
EQNS IDGen1.Next = IDSt(IDGen1) AND IDGen1
ENDCLASS (* IDGen *)
```

```
CLASS NodeID RENAMES ID LITERALS: 0 WITH N0 STRUCTURES: IDSt WITH N
ENDCLASS (* --------------------- NodeID --------------------- *)
CLASS NodeIDGen USING NodeID EXTENDS NodeID WITH OPNS DUALS: NextN -> NodeID
EQNS NodeIDGen1.NextN = N(NodeIDGen1) AND NodeIDGen1
ENDCLASS (* NodeIDGen *)
```

This simple example illustrates the limited use of the `RENAMES` facility. It is not possible to rename both `ID` and `Gen` to create `NodeID` and `NodeIDGen` because the subsequent subclassing relationship between these two classes will not be properly defined. Consequently, it is necessary to define `NodeID` as a renaming of `ID`, and `NodeIDGen` as an extension of `NodeIDGen`[7]. `LinkIDGen` and `LinkID` are also defined similarly: the `STRUCTURE` operation is renamed `L`, the `LITERAL` is renamed `L0` and the `DUAL` is renamed `NextL`.

Step 3.4: Analysis of the NodeSet Class

`NodeSet` is required, by the `Network`, to offer the following external atributes:

- `TRANSFORMER: add< NodeID, NodeType >`
  Create a node from identifier and type components, and add it to the `NodeSet`.
- `TRANSFORMER: switch< NodeID >`
  Search the node set for the node identified by the `NodeID` parameter and switch the state of this node from `on` to `off` (or vice versa).
- `ACCESSOR: isNode< NodeID > -> Bool`
  Return `true` if there is a node in the network which is identified by `NodeID`.
- `ACCESSOR: isOn< NodeID > -> Bool`
  Return true if there is a node in the network identifed by `NodeID` which is `on`, otherwise return false if the identified node is `off`.

The `NodeSet` class definition is given below.

[7] An obvious extension to OO ACT ONE is to provide a more powerful renaming facility which 'copies' class hierarchies rather than individual classes. The investigation of the semantics of such a copy is beyond the scope of this thesis.

```
CLASS NodeSet USING Node OPNS ...(* Operations as specified above *)
EQNS emptyNodeSet..isNode(NodeID1) = false;
(Node1.getID).eq(NodeID1) =>
NodeStr(Node1, NodeSet1)..isNode(NodeID1) = true OTHERWISE NodeSet1..isNode(NodeID1);
emptyNodeSet..getNode(NodeID1) = ~Bool; emptyNodeSet..switch(NodeID1)= ~NodeSet;
(Node1.getID).eq(NodeID1) => NodeStr(Node1, NodeSet1)..getNode(NodeID1) =
  Node1 OTHERWISE NodeSet1..isOn(NodeID1);
NodeSet1.add(NodeID1,NodeType1) = NodeSetStr(ANode(NodeID1,NodeType1),NodeSet1);
(Node1.getID).eq(NodeID1) => NodeSetStr(Node1, NodeSet1)..switch(NodeID1) =
  NodeSetStr(Node1..switch, NodeSet1) OTHERWISE NodeSet1.isNode(NodeID1) =>
    NodeSetStr(Node1, NodeSet1.switch(NodeID1)) OTHERWISE ~NodeSet
ENDCLASS (* NodeSet *)
```

Step 3.5: Analysis of the Node Class

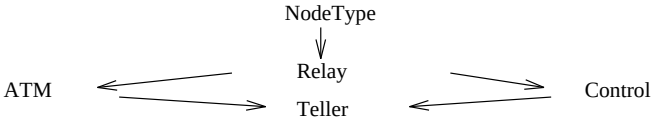
The composition of the `Node` class has already been identified as a fixed `STRUCTURE` (`ANode`) of two components: a `NodeID` and a `NodeType`. The `NodeSet` class places requirements on the external interface of `Node` which are fulfilled by the OO ACT ONE specification given below.

```
CLASS Node Using NodeID, NodeType
STRUCTURES: ANode<NodeID, NodeType> (*FIXED*)
ACCESSORS: isOn -> Bool, isControl -> Bool, isATM -> Bool, getID -> NodeID
TRANSFORMERS: switch
EQNS ANode(...)..isOn = NodeType1..isOn; ANode(...)..isATM = NodeType1..isATM;
ANode(...)..isControl = NodeType1..isControl; ANode(...)..getID = NodeID1; ANode(...).switch =
ANode(NodeID1, NodeType1.switch)
ENDCLASS (* Node *)
```

The `isOn`, `isControl`, `isATM` and `switch` service requests are 'passed on' to the `NodeType` component. The `getId` attribute returns the node identifier. The `NodeID` class has already been specified and so we now consider the `NodeType` class.

Step 3.5.1 Analysis of the NodeType class

The `NodeType` class has been identified as the root of a subclass hierarchy:



At the analysis stage of this case study, the precise role of the `NodeType` classes is not specified. The only requirements placed on `NodeType` objects is that they can be either `on` or `of`, and offer `ACCESSOR` attributes `isOn`, `isControl` and `isATM`. The `NodeType` class (and its subclasses) are specified in OO ACT ONE, below.

```
CLASS NodeType Using Bool OPNS
LITERALS: on, off
ACCESSORS: isOn -> Bool
EQNS on..isOn = true; off..isOn = false ENDCLASS (* NodeType *)
CLASS Relay USING NodeType EXTENDS NodeType WITH OPNS TRANSFORMERS: switch
EQNS on.switch = off; off.switch = on ENDCLASS (* Relay *)
CLASS ATM USING Relay EXTENDS Relay WITH OPNS ACCESSORS: isATM -> Bool
EQNS ATM1..isATM = true ENDCLASS (* ATM *)
CLASS Teller USING ATM EXTENDS ATM WITH OPNS ACCESSORS: isControl -> Bool, isTeller -> Bool
EQNS Teller..isControl = true; Teller..isTeller = true ENDCLASS (* Teller *)
CLASS Control USING Teller RESTRICTS Teller TO OPNS
ACCESSORS: isControl TRANSFORMERS: switch
ENDCLASS (* Control *)
```

Step 3.6 Analysis of LinkSet Class

The `LinkSet` class has been identified as recursive structure of `Links`, with `ACCESSOR` attribute `areConnected` and `TRANSFORMER` attribute `link`. The behaviour of `LinkSet` is formally defined below.

```
CLASS LinkSet Using Link OPNS
LITERALS: emptyLinkSet STRUCTURES: LinkStr<Link, LinkSet>
ACCESSORS: areConnected< NodeID, NodeID > -> Bool
TRANSFORMERS: link < NodeID, NodeID >
EQNS LinkSet1.link(NodeID1, NodeID2, LinkID) = LinkStr(ALink(NodeID1,NodeID2,LinkID), LinkSet1);
empty..areConnected(NodeID1, NodeID2) = false;
(Link1..conn1)..eq(NodeID1)).and((Link1..conn2)..eq(NodeID2)).or(
(Link1..conn2)..eq(NodeID1)).and((Link1..conn1)..eq(NodeID2))) =>
LinkStr(Link1,LinkSet1)..areConnected(NodeID1,NodeID2)= true OTHERWISE
LinkSet1..areConnected(NodeID1,NodeID2)
ENDCLASS (* LinkSet *)
```

The `Link` class is a simple passive holder of data in a fixed structure. It has three external attributes for accessing the values of each of its three components. The OO ACT ONE specification of `Link` is given below.

```
CLASS Link Using LinkID, NodeID OPNS
STRUCTURES: ALink<NodeID, NodeID, LinkID> (*FIXED*)
ACCESSORS: conn1 -> NodeID, conn2 -> NodeID, getID -> LinkID
EQNS ALink(...)..conn1 = NodeID1; ALink(...)..conn2 = NodeID2; ALink(...)..getID = LinkID1
ENDCLASS (* Link *)
```

Step 3.7: *Customer Validation* of Network

The OO ACT ONE specification of the `Network` class is now put forward for *customer validation*. The executable ACT ONE `Network` model is successfully generated and this is used to test the `Network` requirements model. *Customer validation* of the `Network` resulted in two changes being made to the `Network` requirements model. Firstly, an invariant property was added to the `Link`

class to specify that a link cannot connect a node to itself: `ALink(NodeID1, NodeID2, LinkID1) REQUIRES (NodeID1..eq(NodeID2)).not`. Secondly, in order to ensure the correct addition of links to the network, an exception was defined to occur when a request is made to connect a node to itself: `NodeID1.eq(NodeID2) => Network1.addLink(NodeID1, NodeID2) = ~Network OTHERWISE (* as before *)`.

The process of validation brought the *quality* of the requirements model into question:

- **Question:** Could `NodeSet` and `LinkSet` be *better* defined as instances of a generic `Set`?
  Answer: The `NodeSet` and `LinkSet` classes provide quite specific behaviour: different types of access to (and transformation of) individual set elements. A standard generic set definition is not suitable for the parameterisation of these two behaviours.
- **Question:** Could the `Network` behaviour be defined, for re-use, in a generic class?
  Answer: Generic network behaviour can be usefully defined: graphs of nodes and links are common in computing systems. This is noted for future investigation and development.
- **Question:** Is the *customer view* of the network as having separate `NodeSet` and `LinkSet` components the *best* way of conceptualising (and communicating) the requirements?
  Answer: As the requirements model was developed and validated, a better understanding of the `Network` evolved. It was felt that a better understanding of the requirements could have been achieved if the nodes in the network had been specified as 'knowing' the other nodes to which they were connected, rather than having a separate `LinkSet`. This type of reconceptualisation must always be proposed to the *customer*: only after they agree that the new model is a *better* recording of their understanding of the requirements can appropriate changes be made. In the case study, for reasons explained at the beginning of this chapter, the process of *customer validation* could not be properly evaluated, and the process of reconceptualisation was not carried out. Examination of the process of communication between customer and analyst, particularly the influence of the analyst on the way in which the customers conceptualise their requirements, is beyond the scope of this thesis.

Step 3.7: Provide a Graphical View of Network State

The OO ACT ONE requirements model provides an excellent statement of `Network` behaviour, which can be supplemented by graphical views. It is often advantageous (particularly for complex classes of behaviour) to record, in the requirements documentation, the correspondence between:

- The OO ACT ONE representation of a class member.
- The structure diagram of a class member.
- The customer's conceptualisation of a class member.

When it is clear that the customer's view of the requirements provides a useful way of representing the behaviour of the system then an attempt should be made to provide a formal semantics, founded on the OO ACT ONE specification, of their representation. An example of this is given in figure 7.3, where the customer's view of a particular network is given a formal meaning.

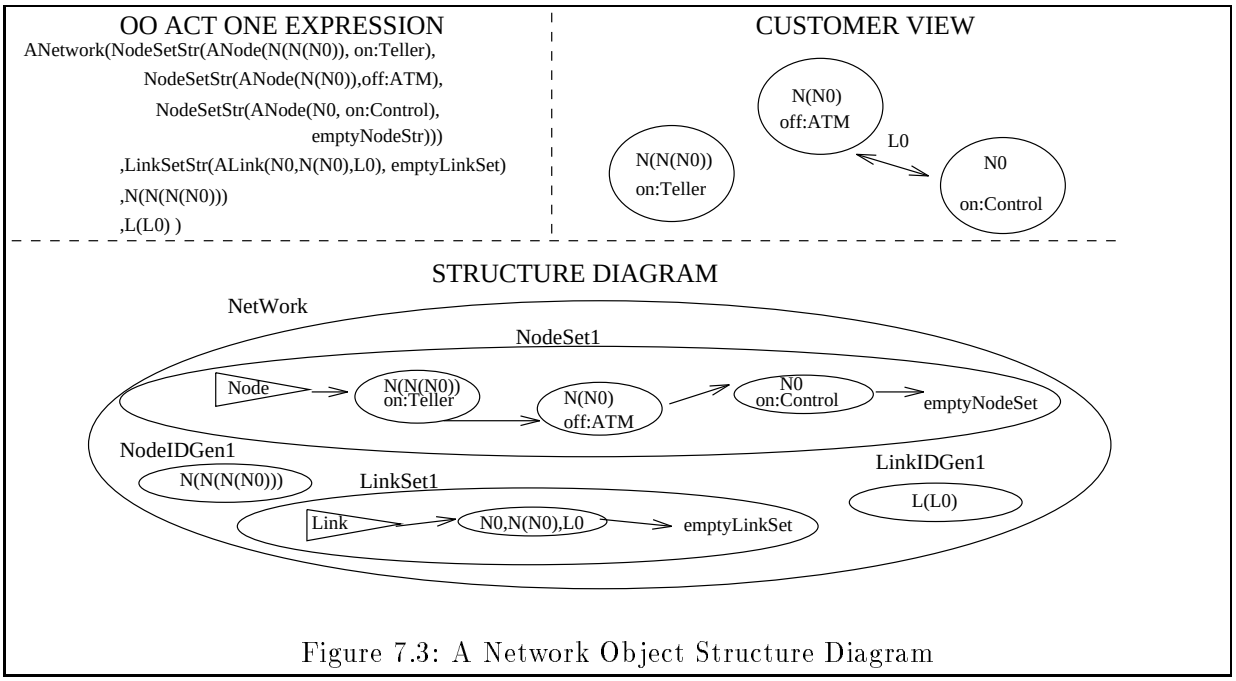


Figure 7.3: A Network Object Structure Diagram

Step 3.7: Make The Network Available For Design

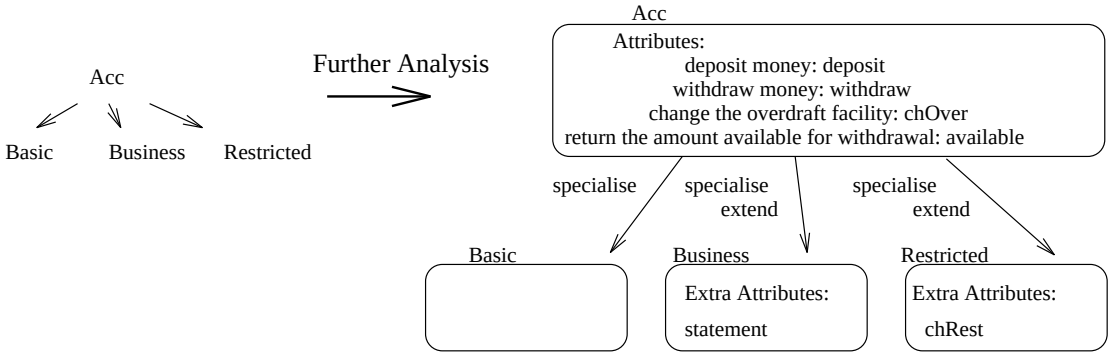
After *customer validation*, the `Network` is ready for design and implementation. The decision to proceed with its development must be taken by the project managers. The risk of developing the `Network` before the analysis of the `BankingNetwork` is complete must be weighed against the advantages of running design and implementation in parallel with the unfinished analysis and requirements process. This risk must be evaluated for every class in the system being analysed.

Step 4: Analysis of the AccDB Class and Synthesis of the Requirements Model

After the `Network`, the account database (`AccDB`) appears to play the next most significant role in the behaviour of the banking network. The `AccDB` class is a recursive structure of `Acc` classes. Before we analyse `AccDB` behaviour we analyse the behaviour of its `Acc` components.

Step 4.1 Analysis of Acc Behaviour

It is useful to analyse the relationship between `Acc` and each of its subclasses, namely `Basic`, `Business` and `Restricted`. Further analysis of the informal requirements leads to a more informative class hierarchy:



Every account has identifier, balance and overdraft components. Restricted accounts have a restriction on the amount of money which can be withdrawn in one transaction. Business accounts can supply, on request, a statement of the last three transactions which the account has processed. The balance, overdraft and restriction components are defined to belong to the class `Sum`, which represents an amount of money (positive or negative).

As a result of this analysis, three new classes are identified:

- `AccID`, which defines a means of uniquely identifying accounts. This is defined as a renaming of `ID`: the `LITERAL 0` is renamed `A0`, the `STRUCTURE IDSt` is renamed `A` and the `DUAL Next` is renamed `NextA`.
- `Trans3X`, which defines a record of the last three transactions that a business account has serviced. This class of behaviour is synthesised and analysed in step 4.3.
- `Sum` provides a means of recording positive and negative amounts of money. It also provides the necessary arithmetic for the manipulation and testing of these amounts. The class `Sum` can be defined as a renaming of some standard class of *numbers*[8].

Step 4.1.1 An initial `Acc` model

A sequence of OO ACT ONE `Acc` models were developed to improve understanding of the requirements. The first such model is defined below (as version[9] 1).

Analysis of version 1 of the `Acc` requirements model gives rise to a number of questions which must be answered by the *customer*:

- i) What happens when a withdrawal is requested which is greater than the amount available?
- ii) What happens when a withdrawal is requested of a restricted account which exceeds the restriction?
- iii) What happens when a change of the overdraft facility is requested which would result in the overdraft being exceeded?

[8] Like ACT ONE, OO ACT ONE is not well suited to representing numerical behaviour. The definition of *numbers* is not reported in this thesis.

[9] Often it is necessary to construct a prototype model (or models) as a means of improving the mutual understanding between customer and analyst. These prototypes must be clearly distinguished from the final requirements models: it is recommended that every prototype is given a version number and the documentation include details of what was learned from the analysis of each version.

```
CLASS Acc Using AccID, Sum, Trans3X OPNS (* Version 1 *)
STRUCTURES: BasicStr<AccID, Sum (*balance*), Sum (*overdraft*) >,
BusinessStr<AccID, Sum (*balance*), Sum (*overdraft*), Trans3X>,
RestStr<AccID, Sum (*balance*), Sum (*overdraft*), Sum (*restriction*)>
ACCESSORS: available -> Sum, getID -> AccID
TRANSFORMERS: deposit<Sum>, withdraw<Sum>, chOver<Sum>
EQNS BasicStr(AccID1,Sum1,Sum2)..available = Sum1.add(Sum2);
BasicStr(AccID1,Sum1,Sum2)..getID = AccID1;
BasicStr(AccID1,Sum1,Sum2).chOver(Sum3) = BasicStr(AccID1,Sum1,Sum3);
(* available, getID and chOver are defined similarly for the other structures *)
BasicStr(AccID1,Sum1,Sum2).deposit(Sum3) = BasicStr(AccID1,Sum1.add(Sum3),Sum2);
BasicStr(AccID1,Sum1,Sum2).withdraw(Sum3) = BasicStr(AccID1,Sum1.sub(Sum3),Sum2);
BusinessStr(AccID1,Sum1,Sum2,Trans3X1).deposit(Sum3) =
  BusinessStr(AccID1,Sum1.add(Sum3),Sum2,Trans3X1.insert(ATrans(DepositStr(Sum3))));
BusinessStr(AccID1,Sum1,Sum2,Trans3X1).withdraw(Sum3) =
  BusinessStr(AccID1,Sum1.sub(Sum3),Sum2,Trans3X1.insert(ATrans(WithdrawStr(Sum3)))));
RestStr(AccID1,Sum1,Sum2,Sum3).deposit(Sum4) = RestStr(AccID1,Sum1.add(Sum4),Sum2,Sum3);
RestStr(AccID1,Sum1,Sum2,Sum3).withdraw(Sum4) = RestStr(AccID1,Sum1.sub(Sum4),Sum2,Sum3);
ENDCLASS (* Acc *)
```

- iv) Can the overdraft, restriction, deposits and withdrawals be negative? If so, what happens in each case?

Step 4.1.2: Backtracking

The analysis also identifed a misrepresentation in the way in which the `available` attribute is defined for restricted accounts: the restriction amount should be returned when this is smaller than the sum of the balance and overdraft amounts. This misrepresentation was corrected by making a change in the final version of the `Acc` requirements model. The `Acc` model was also changed to record the following additional requirements:

- All the scenarios in questions (i) to (iii) result in exceptions which must be dealt with by designers and/or implementers.
- A negative overdraft is used to model a minimum amount that must be kept in an account.
- A negative restriction is not permitted by the definition of an invariant property, and a request to change a restriction to a negative amount results in an exception.
- Requests to deposit or withdraw negative amounts are also defined as exception cases.
- Additional `ACCESSOR` attributes are defined to test the type of a given account.
- An additional `ACCESSOR zeroBalance` is defined to return true if the balance of an account is zero. This test is needed for closing accounts.

Step 4.1.3 A Final `Acc` model

The new version of the `Acc` requirements is re-tested and *customer validated*. This final version of the model is defined below.

```
CLASS Acc Using AccID,TransResult,Sum,Trans3X (*FINAL*) STRUCTURES: (*As in version 1*)
ACCESSORS: available -> Sum, getID -> AccID,
zeroBalance -> Bool, isBasic -> Bool, isBusiness -> Bool, isRest -> Bool;
INVARIANTS: Reststr(AccID1,Sum1,Sum2,Sum3) REQUIRES Sum3..positive
EQNS (* available, getID and chOver:  as defined in version 1 *)
BasicStr(AccID1, Sum1, Sum2)..isBasic = true; (* other structures defined similarly *)
BasicStr(AccID1, Sum1, Sum2)..isBusiness = false; (* other structures defined similarly *)
BasicStr(AccID1, Sum1, Sum2)..isRest= false; (* other structures defined similarly *)
BasicStr(AccID1, Sum1, Sum2)..zeroBalance= Sum1.eq(0); (* other structures defined similarly *)
Sum3..positive => BasicStr(AccID1, Sum1, Sum2).deposit(Sum3) =
  BasicStr(AccID1, Sum1.add(Sum3), Sum2) OTHERWISE ~Acc;
Sum3..positive => BasicStr(AccID1, Sum1, Sum2).withdraw(Sum3) =
BasicStr(AccID1, Sum1.sub(Sum3), Sum2) OTHERWISE ~Acc;
Sum3..positive => BusinessStr(AccID1, Sum1, Sum2, Trans3X1).deposit(Sum3) =
  BusinessStr(AccID1,Sum1.add(Sum3),Sum2,Trans3X1.insert(ATrans(DepositStr(Sum3))))
  OTHERWISE ~Acc Sum3..positive => BusinessStr(AccID1,Sum1,Sum2,Trans3X1).withdraw(Sum3) =
  BusinessStr(AccID1,Sum1.sub(Sum3),Sum2,Trans3X1.insert(ATrans(WithdrawStr(Sum3))))
  OTHERWISE ~Acc;
Sum4..positive => RestStr(AccID1,Sum1,Sum2,Sum3).deposit(Sum4) =
RestStr(AccID1,Sum1.add(Sum4),Sum2,Sum3) OTHERWISE ~Acc;
Sum4..positive => RestStr(AccID1,Sum1,Sum2,Sum3).withdraw(Sum4) =
RestStr(AccID1,Sum1.sub(Sum4),Sum2,Sum3) OTHERWISE ~Acc ENDCLASS (* Acc *)
```

Step 4.2 Analysis and Synthesis of `Acc` Subclasses

The `Basic` class is defined as a specialisation of `Acc`, restricted to values represented by the `BasicStr STRUCTURE`. The `Business` and `Restricted` classes are defined to specialise and extend `Acc`. These classes are defined below. Their behaviours, being based on the already validated `Acc` class, are easy to check with the customer. An important point to note is that `Business` and `Restricted` are not defined as subclasses of `Basic`. These classes could have been defined in this way but it was clear that the three different types of account were intended to be unrelated: for example, a business account is not a basic account.

```
CLASS Basic USING Acc SPECIALISES ACC TO OPNS STRUCTURES: BasicStr
ENDCLASS (* Basic *)
CLASS Business USING Acc SPECIALISES AND EXTENDS ACC TO OPNS
STRUCTURES: BusinessStr ACCESSORS: statement -> Trans3X
EQNS BusinessStr(AccID1, Sum1, Sum2, Trans3X1)..statement= Trans3X1
ENDCLASS (* Business *)
CLASS Restricted USING Acc SPECIALISES AND EXTENDS ACC TO OPNS
STRUCTURES: RestStr TRANSFORMERS: chRest<Sum>
EQNS Sum4..positive => RestStr(AccID1, Sum1, Sum2, Sum3).chRest(Sum4) =
  RestStr(AccID1, Sum1, Sum2, Sum4) OTHERWISE ~RestStr
ENDCLASS (* Restricted *)
```

Step 4.3: Analysis of Trans3X Class

This class is required to record the last three transactions which have been serviced on a business account. A transaction is added using the `insert` attribute. The most recent transaction is stored as the first component, and the least recent as the third component, of a fixed `STRUCTURE`. This behaviour is specified below.

```
CLASS Trans3X USING TrReq OPNS
STRUCTURES: ATrans3X< TrReq, TrReq, TrReq > (*FIXED*)
TRANSFORMERS: insert<TrReq>
EQNS ATrans3X(TrReq1, TrReq2, TrReq3).insert(TrReq4) = ATrans3X(TrReq4, TrReq1, TrReq2)
ENDCLASS (* Trans3X*)
```

Step 4.4: Identifying an Account Transaction Hierarchy

The different types of account transactions are represented in the class hierarchy in figure 7.4.

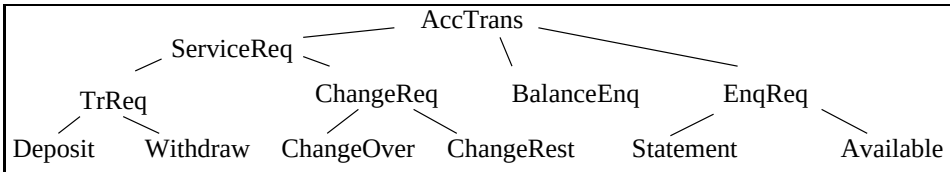


Figure 7.4: The Account Transaction Class Hierarchy

The most important class in this hierarchy is `ServiceReq`. This is used to parameterise the `TRANSFORMER` attributes on the account database, and consequently simplify the specification. All the classes in the account transaction hierarchy are defined, in OO ACT ONE, below.

Step 4.5: Analysis and Synthesis of AccDB

The account database must provide a store for accounts and a means of accessing and updating the information associated with each account in the store. The analysis and synthesis of the account database class `AccDB` required many iterations of the analysis-synthesis-backtrack sequence. The header for the final version of the `AccDB` class is specified below.

All `AccDB TRANSFORMER` services are offered through one external attribute, namely `service`. The attribute `service` is then parameterised on an account identifier and transformer transaction (of class `ServiceReq`. This parameterisation helps to simplify the external interface of the `AccDB` class. Another important aspect of the `AccDB` class is that exceptions are specified when services are requested of accounts which are not in the account database. The hidden attribute `serviceOK`, used by the external attribute `service`, is defined (and called) only for services on valid accounts. This behaviour is specified by the equation definitions of the `AccDB` class, below.

An important aspect of the `AccDB` specification is the way in which changes to particular accounts in the database are made. The `TRANSFORMER` operations are defined by removing the identified account,

```
CLASS TrReq USING Sum OPNS STRUCTURES: DepositStr<Sum>,WithdrawStr<Sum> ENDCLASS
CLASS EnqReq OPNS LITERALS: StatementLit, AvailableLit ENDCLASS (*TrReq*)
CLASS BalanceEnq OPNS LITERALS: BalanceLit
ACCESSORS: isService -> Bool,isStatement -> Bool,isAvailable -> Bool ENDCLASS (*BalanceEnq*)
CLASS ChangeReq OPNS STRUCTURES: ChOverStr<Sum>, ChangeReqStr<Sum> ENDCLASS(* ChangeReq*)
CLASS Deposit USING TrReq SPECIALISES TrReq TO OPNS STRUCTURES: DepositStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Deposit*)
CLASS Withdraw USING TrReq SPECIALISES TrReq TO OPNS STRUCTURES: WithdrawStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Withdraw*)
CLASS Statement USING EndReq SPECIALISES EndReq TO OPNS LITERALS: StatementLit
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Statement*)
CLASS Available USING EndReq SPECIALISES EndReq TO OPNS LITERALS: AvailableLit
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*Available*)
CLASS ChOver USING ChangeReq SPECIALISES ChangeReq TO OPNS STRUCTURES: ChOverStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*ChOver*)
CLASS ChRest USING ChangeReq SPECIALISES ChangeReq TO OPNS STRUCTURES: ChRestStr
ACCESSORS: isService -> Bool, isStatement -> Bool, isAvailable -> Bool ENDCLASS (*ChRest*)
CLASS ServiceReq USING TrReq, ChangeReq GENERALISES TrReq, ChangeReq ENDCLASS (*ServiceReq*)
CLASS AccTrans USING ServiceReq,BalanceReq,ChangeReq
GENERALISES ServiceReq,BalanceReq,ChangeReq ENDCLASS (*AccTrans*)
(* The ACCESSOR equations are not given above:  their definitions are intuitive *)
```

```
CLASS AccDB USING Acc, ServiceReq OPNS
LITERALS: emptyAccDB STRUCTURES: AccStr< Acc, AccDB >
ACCESSORS:available<AccID> -> Sum,statement<AccID> -> Trans3X,
getAcc<AccID> ->Acc(*HIDDEN*),isAcc<AccID>
->Bool(*HIDDEN*),zeroBalance<AccID>->Bool(*HIDDEN*)
TRANSFORMERS: newAcc<Acc>,delAcc<AccID>,service<AccID,ServiceReq>,
deposit<AccID,Sum> (*HIDDEN*), withdraw<AccID,Sum> (*HIDDEN*),
chOver<AccID,Sum>(*HIDDEN*),chRest<AccID,Sum>(*HIDDEN*),serviceOK<AccID,ServiceReq>(*HIDDEN*)
EQNS ...ENDCLASS (* AccDB *)
```

updating the account in an appropriate fashion and reconstructing the database by adding in the old account with its new state. The behaviour defined in this way is simple to understand but it should be clear that it is definitely not efficient. The analyst *must not* change the requirements model just because it is inefficient: the most important property of the requirements model is that the customer can understand it.

Step 5: Analysis of Transaction Class and Synthesis of Requirements Model

The `Transactions` class is used to store the account transaction messages, of class `TransMessage`, which are currently being routed to/from the target control node. It is defined as a recursive structure of `TransMessage` elements. It has an external interface composed from two attributes: a `TRANSFORMER add` and a `DUAL remove`. The class is defined below.

The behaviour of the `Transactions` class is simple to understand and easy to communicate with the customer. Consequently, *customer validation* of its behaviour is straightforward.

```
CLASS AccDB ...(* Header as above *) EQNS emptyAccDB.getAcc(AccID1) = ~Acc;
emptyAccDB.isAcc(AccID1) = false;
emptyAccDB.zeroBalance(AccID1) = ~Bool; emptyAccDB.available(AccID1) = ~Sum;
emptyAccDB.delAcc(AccID1) = ~AccDB;
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).getAcc(AccID1) = Acc1
OTHERWISE AccDB1.getAcc(AccID1);
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).isAcc(AccID1) = true
OTHERWISE AccDB1.isAcc(AccID1);
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).zeroBalance(AccID1) = Acc1.zeroBalance
OTHERWISE AccDB1.zeroBalance(AccID1);
(Acc1.getID).eq(AccID1) => AccStr(Acc1, AccDB1).available(AccID1) = Acc1.available
OTHERWISE AccDB1.available(AccID1);
AccDB1.getAcc(AccID1)..isBusiness => AccDB1..statement(AccID1) =
  AccDB1.getAcc(AccID1)..statement OTHERWISE ~Trans3X;
AccDB1.newAcc(Acc1) = AccStr(Acc1,AccDB1);
(Acc1.getID).eq(AccID1)).not => AccStr(Acc1, AccDB1).delAcc(AccID1)) =
  AccStr(Acc1, AccDB1.del(AccID1)) OTHERWISE (Acc1..balance).eq(0) => AccDB1 OTHERWISE ~AccDB;
AccDB1.getAcc(AccID1)..isRest => AccDB1..chRest(Acc, Sum1) =
  (AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..chRest(Sum1)) OTHERWISE ~AccDB1;
AccDB1.deposit(AccID1, Sum1) =
(AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..deposit(Sum1));
AccDB1.withdraw(AccID1, Sum1) =
(AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..withdraw(Sum1));
AccDB1.chOver(AccID1, Sum1) =
(AccDB1.delAcc(AccID1)).newAcc(AccDB1..getAcc(AccID1)..chOver(Sum1));
AccDB1.isAcc(AccID1) => AccDB1.service(AccID1, ServiceReq1) =
  AccDB1.serviceOK(AccID1, ServiceReq1) OTHERWISE ~Acc;
AccDB1.serviceOK(AccID1, ChOverStr(Sum1)) = AccDB1.chOver(AccID1, Sum1);
AccDB1.serviceOK(AccID1, ChRestStr(Sum1)) = AccDB1.chRest(AccID1, Sum1);
AccDB1.serviceOK(AccID1, DepositStr(Sum1)) = AccDB1.deposit(AccID1, Sum1);
AccDB1.serviceOK(AccID1, WithdrawStr(Sum1)) = AccDB1.withdraw(AccID1, Sum1);
ENDCLASS (* AccDB *)
```

```
CLASS Transactions USING TransMessage OPNS
LITERALS: NoTrans STRUCTURES: TransStr< Trans, Transactions >
TRANSFORMERS: add<TransMessage> DUALS: remove<MessageID> -> TransMessage
EQNS Transactions1.add(TransMessage1) = TransStr(TransMessage1, Transactions1);
NoTrans.remove(MessageID1) = NoTrans;
(TransMessage1.getID).eq(MessageID1) =>
TransStr(TransMessage1, Transactions1).remove(MessageID1) = Transactions AND TransMessage1
OTHERWISE TransStr(TransMessage1,Transactions1.remove(MessageID1)
ENDCLASS (* Transactions *)
```

Step 5.1: Analysis of TransMessage Class and Synthesis of Requirements Model

The `TransMessage` class is defined as a fixed `STRUCTURE` of five components:

- A message identifier, which is defined as a renaming of `ID`. The `LITERAL 0` is renamed `M0` and the `STRUCTURE IDStr` is renamed `M`.

- An identification of the target control node, defined as a `NodeID`.
- An identification of the node which originated the transaction request, defined as a `NodeID`.
- An identification of the account to which the request is being sent, defined as an `AccID`.
- The request details, defined as an `AccTrans`.

The `TransMessage` class offers three external attributes:

- an `ACCESSOR getAccTrans` to return the `AccTrans` component.
- an `ACCESSOR getAccID` to return the `AccID` component.
- an `ACCESSOR getMessageID` to return the `MessageID` component.

The behaviour of class `TransMessage` is specified below.

```
CLASS TransMessage USING AccTrans, NodeID, MessageID OPNS
STRUCTURES: ATransMessage< MessageID, NodeID (*to*), NodeID (*from*), AccID, TrReq >
ACCESSORS: getMessageID ->MessageID,getAccTrans ->AccTrans, getAccID ->AccID
EQNS ATransMessage(MessageID1,NodeID1,NodeID2,AccID1,AccTrans1)..getMessageID = MessageID1;
ATransMessage(MessageID1,NodeID1,NodeID2,AccID1,AccTrans1)..getAccID = AccID1;
ATransMessage(MessageID1,NodeID1,NodeID2,AccID1,AccTrans1)..getAccTrans = AccTrans1
ENDCLASS (* TransMessage *)
```

Step 6: Specifying a class for returning enquiry results

A new class, `AccTransResult`, is required to represent the result of a transaction enquiry at the `BankingNetwork` interface. This class was not initially identifed in the composition analysis because it is not a component of the `BankingNetwork`. However, it is now clear that such a class is required for the transfer of enquiry results across the network. `AccTransResult` is defined below.

```
CLASS AccTransResult USING Sum, Trans3X OPNS
STRUCTURES: StatementRes< Trans3X >, AvailableRes < Sum >
ENDCLASS (* AccTransResult *)
```

`AccTransResult` requires no external attributes because it is being used as a passive data representation. Classes which use the `BankingNetwork` can re-use `AccTransResult` and extend it with external attributes, if necessary.

Step 7: Synthesis of BankingNetwork Class

The classes which `BankingNetwork` depends on have been *customer validated*. The last step in the case study is the synthesis and analysis of the `BankingNetwork` class. Again, like its component classes, the `BankingNetwork` required many iterations of the analysis-synthesis-backtrack sequence before the requirements model was *customer validated*.

Step 7.1 Review of the BankingNetwork Component Classes

The `BankingNetwork` is composed from three component classes:

- `Network`: a network of nodes and links.
- `Transactions`: a set of internal message transactions which are in the process of being routed.
- `AccDB`: the data base of accounts which is the target for all the transactions in the system.

The *Interface O-LSTSD* for each of these classes is given in figure 7.5. The `BankingNetwork` provides its external functionality by delegating requests to these component classes.

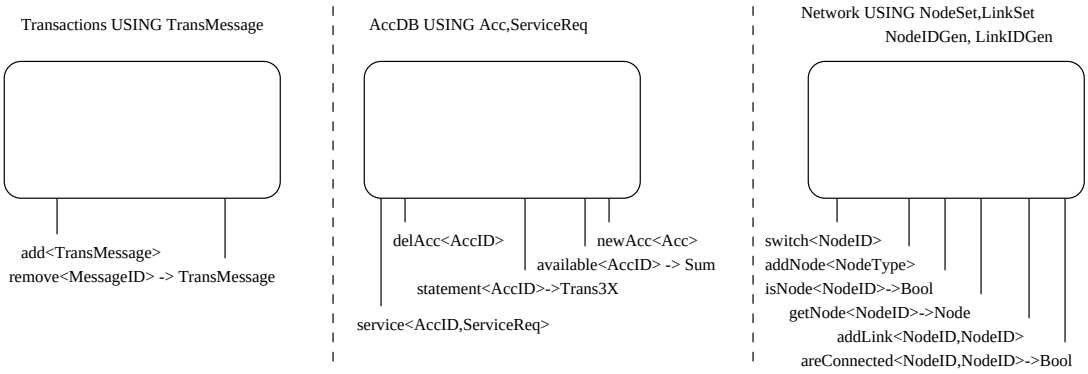


Figure 7.5: A Review of the BankingNetwork Components

Step 7.2 The External Interface of the BankingNetwork

The external interface of the `BankingNetwork` is defined by the class header, below.

```
CLASS BankingNetwork USING AccDB, AccTransResult, Network, Transactions OPNS
STRUCTURES: ABankNet< AccDB, Network, Transactions >
TRANSFORMERS: addNode<NodeID,NodeType>, addLink<NodeID,NodeID,NodeID>, switch<NodeID>,
accRequest<NodeID, TransMessage>, delAcc<NodeID, AccID>, newAcc<NodeID, Acc>,
arrived<MessageID>(*INTERNAL*), timeout<MessageID>(*INTERNAL*)
DUALS: returned<MessageID> -> AccTransResult (*INTERNAL*) EQNS ...
```

There are four important aspects of the `BankingNetwork` header definition:

- i) The `INTERNAL` transformers are used to model the communication aspects of the system:
  - The `arrived` sevice models a message transaction arriving at the appropriate control node, and results the account information being updated.
  - The `returned` service models the arrival of an enquiry reply at the node which originated the enquiry.
  - The `timeout` service models the nonservicing of a message transaction.
- ii) All the external attributes are defined as `TRANSFORMERS`. The enquiry services cannot be defined as `DUALS` because there is no guarantee that these requests will be serviced.

- iii) All the external attributes have a `NodeID` as their first parameter which is used to identify the node at which a transaction is originated. The `BankingNetwork` must ensure that only the correct sort of transactions are requested at particular types of node.
- iv) The `switch` attribute is interesting because the one `NodeID` parameter identifies both the node to be switched and the node at which the `switch` request is being made. In other words, `switch` requests cannot be routed across the network: they must occur at source.

The `BankingNetwork` class structure diagram is given in figure 7.6.

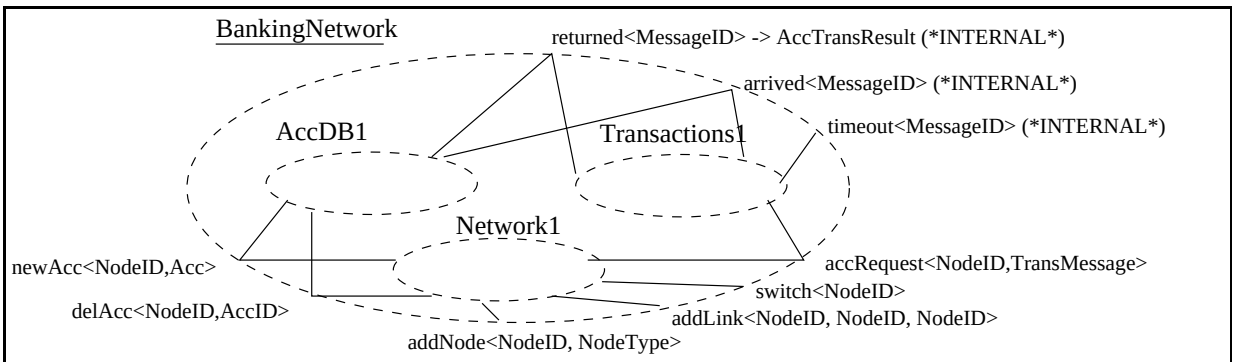


Figure 7.6: The BankingNetwork Class Structure Diagram

Step 7.3 Specification of BankingNetwork Behaviour

Informally, the external attributes of the `BankingNetwork` offer the following behaviour:

- `addNode<NodeID, NodeType>`:
  The node identifier is checked, using the `Network` component, to guarantee that it corresponds to a control node in the system which is on: if so, a new node of type `NodeType` is added to the `Network`, otherwise the system ignores the request.
- `addLink<NodeID, NodeID, NodeID>`:
  All three node identifiers are checked, using the `Network` component, to test that:
  - the first node identifier corresponds to a control node which is on.
  - the other two node identifiers correspond to different nodes in the `Network`

  If so, a new link, joining the nodes identified by the second two `NodeID` parameters, is added to the `Network`, otherwise the system ignores the request.
- `newAcc<NodeID,Acc>`:
  The node identifier is checked, using the `Network` component, to guarantee that it corresponds to a control node in the system which is on: if so, a new account (`Acc`) is added to the `AccDB`, otherwise the system ignores the request.
- `delAcc<NodeID, AccID>`:
  The node identifier is checked, using the `Network` component, to guarantee that it corresponds

to a control node in the system which is on: if so, the account is checked to ensure that it has a zero balance, in which case it is deleted from the database, otherwise the system ignores the request.

- `switch<NodeID>`:
  The `NodeID` is tested to verify that it corresponds to a node in the `Network`: if so, this node is switched, otherwise the system ignores the request.
- `accRequest<NodeID, TransMessage>`:
  The `NodeID` is tested to verify that it corresponds to an ATM node which is on: if so, the `TransMessage` request is added to the system `Transactions` component for routing to the appropriate control node, otherwise the system ignores the request.

The `INTERNAL` attributes model nondeterministic state transitions:

- `timeout<MessageID>`:
  This removes the specified message from the `Transactions` component of the `BankingNetwork`.
- `arrived<MessageID>`:
  This removes the specified message from the `Transactions` component of the `BankingNetwork`. The message is then used to update the account database `AccDB`. The request is ignored if the message identifier does not correspond to a service request (deposit or withdraw).
- `returned<MessageID>`:
  This removes the specified message from the `Transactions` component of the `BankingNetwork`. The message is then used to access the relevant account information in the account database `AccDB`, and this data is returned. The request is ignored if the message identifier does not correspond to an enquiry request (statement or available).

The `BankingNetwork` behaviour is formally defined in OO ACT ONE below.
The `BankingNetwork` is now ready for design.

7.2.3 A Review of the Analysis and Requirements Capture

The formal object oriented analysis, as performed in the case study, illustrates many of the main analysis and requirements capture issues:

- **Flexibility**: The need for an opportunistic method.
- **Executability**: The advantages of an executable model.
- **Customer Orientation**: The advantages of graphical notations.
- **Formality:** Abstraction and Nondeterminism.

Each of these issues is reviewed, with respect to the case study, in sections 7.2.3.1 to 7.2.3.4, below.

```
CLASS BankingNetwork (* Header above *) EQNS
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).addNode(NodeID1,NodeType1) =
  ABankNet(AccDB1,Network1.addNode(NodeType1),Transactions1) OTHERWISE ABankNet(...);
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn ) =>
ABankNet(...).addLink(NodeID1, NodeID2, NodeID3) =
  ABankNet(AccDB1,Network1.addLink(NodeID2,NodeID3),Transactions1) OTHERWISE ABankNet(...);
ABankNet(...).switch(NodeID1) = ABankNet(AccDB1, Network1.switch(NodeID1), Transactions1);
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).newAcc(NodeID1,Acc) =
  ABankNet(AccDB1.newAcc(Acc),Network1,Transactions1) OTHERWISE ABankNet(...);
(((Network1.getNode(NodeID1))..isControl).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).delAcc(NodeID1,AccID) =
  ABankNet(AccDB1.delAcc(AccID),Network1,Transactions1) OTHERWISE ABankNet(...);
(((Network1.getNode(NodeID1))..isATM).and((Network1.getNode(NodeID1))..isOn) =>
ABankNet(...).accRequest(NodeID1,TransMessage1) =
  ABankNet(AccDB1,Network1,Transactions1.add(TransMessage1)) OTHERWISE ABankNet(...);
ABankNet(...).timeout(NodeID1,MessageID1) =
ABankNet(AccDB1,Network1,Transactions1.remove(MessageID);
((Transactions1..remove(MessageID))..getAccTrans)..isStatement =>
ABankNet(...).returned(NodeID1,MessageID1) =
  ABankNet(AccDB1,Network1,Transactions1.remove(MessageID1)) AND
  StatementRes(AccDB1..statement((Transactions1..remove(MessageID))..getAccID))
  OTHERWISE ((Transactions1..remove(MessageID))..getAccTrans)..isAvailable =>
  ABankNet(AccDB1,Network1,Transactions1.remove(MessageID1)) AND
  AvailableRes(AccDB1..available((Transactions1..remove(MessageID))..getAccID))
  OTHERWISE ABankNet(...);
((Transactions1..remove(MessageID))..getAccTrans)..isService =>
ABankNet(AccDB1,Network1,Transactions1).arrived(MessageID1)=
    ABankNet(AccDB1.service((Transactions..remove(MessageID1))..getAccID,
                            (Transactions..remove(MessageID1))..getAccTrans1),
             Network1,Transactions1.remove(MessageID1)) OTHERWISE ABankNet(...)
ENDCLASS (* BankingNetwork *)
```

7.2.3.1 Flexibility

The case study illustrates how the requirements model was developed in an opportunistic fashion. The analysis was both *bottom-up* and *top-down*:

- *Bottom-up*
  The `BankingNetwork` was analysed *bottom-up*. The classes which the `BankingNetwork` depends on, including its component classes, were specified and validated before the `BankingNetwork` behaviour was fully understood: they were developed to achieve this understanding.
- *Top-down*
  The `Network` behaviour was analysed *top-down*. The classes it depends on were specified and validated after the `Network` behaviour was understood: they were developed to record this

understanding.

In general, it is not possible, given a collection of classes requiring specification, to predetermine the *best* order in which these classes should be analysed, synthesised and validated. Customers and analysts must be encouraged to develop the requirements models as they see fit at the time. Opportunistic approaches are more difficult to organise (the size of the case study does not fully illustrate this) and so it is important that the requirements capture process is supported by management tools. Providing such support is beyond the scope of this thesis.

7.2.3.2 Executability

The OO ACT ONE requirements models are executable. This is very important in the process of *customer validation*. It is very difficult to communicate dynamic properties of a model if the requirements cannot be executed. Executing the requirements models helps the analyst to test their model against the behaviour they think the customer requires. It also helps the customer to validate the analyst's model against their requirements. Further, an executable model helps the customer and analyst to explore behaviour which is not well understood.

The development of the `BankingNetwork` requirements model involved many executions of the system class (and classes which the system depended on). Most of the backtracking took place in response to misunderstandings in the requirements model being identified during execution. It is unlikely that such problems would have been identifed before the system was subsequently implemented. Only after the implementation was tested by the customer would these errors become evident. Executable requirements models reduce the risk of carrying errors into the design and implementation stages of software development.

Executable models are more costly to produce, but have the potential to reduce the expense of correcting errors during design and implementation. Further, the executable OO ACT ONE model is directly re-used in the object oriented designs. Consequently, there is no sense of losing work when the analysis is complete and design begins.

7.2.3.3 Customer Orientation

The OO ACT ONE requirements models are *customer oriented*. The `BankingNetwork` behaviour can be presented to the customer in a number of different forms:

- The dynamic model can be presented as sequences of state transitions. Event diagrams (see 3.5.6) provide a *customer oriented* view of system executions.
- The static properties, namely classification, subclassing, composition, dependency and configuration, can be presented in class structure diagrams and subclassing hierarchy diagrams. These graphical views were prominent in the `BankingNetwork` case study.
- The object oriented framework provides a consistent approach for the analyst to develop mutual understanding with the customer. The customer does not have to change conceptual frameworks when going between their understanding of the requirements and their understanding of the

requirements model: the object oriented style of specification means that these *should* be the same. Analysts must build the requirements model for the customer. This *customer awareness* is evident in the `BankingNetwork`: for example, we questioned the way in which the `Network` behaviour was specified.

Although the graphical views of the requirements were prominent in the case study, it became clear that the synthesis, analysis and validation of behaviour would benefit greatly from comprehensive tool support: all models and views were generated by hand. The ability to make quick changes to a requirements model was not matched by a similarly quick means of presenting the graphical views. This is an area of further work.

7.2.3.4 Abstraction and Nondeterminsim

Two main types of abstraction are evident in the formal requirements model:

- Functional abstraction — every class of behaviour can be treated as an interface of well-defined (and well understood) external attributes.
- Exceptions — when a customer wants some sort of behaviour to be coped with in the final implementation, but is not yet willing (or able) to be more precise then exceptions are useful abstraction mechanisms.

Nondeterminism is a very important part of most requirements models. In the case study nondeterminism (in the form of `INTERNAL` state transitions) was used to model the internal routing of messages. Some messages are routed correctly whilst others are lost (timed out): *how* this behaviour is defined is not specified. It is the role of the analyst to communicate the nondeterministic aspects of the requirements model with the designers. It is the designers who must remove the nondeterminism.

7.3 Design: Moving the System from Abstract to Concrete

This section reviews the process by which the OO ACT ONE `BankingNetwork` requirements were designed for implementation in Eiffel. Design proceeded in distinct steps. At the end of each step the new design was verified against the old design: when a correctness preserving transformation (CPT) was used then this verification was immediate. However, some of the design steps were not applications of pre-defined CPTs. In these cases, we either: formally verified the design step with a proof of correctness, or informally justified the design step as being correctness preserving and tested our reasoning by executing the LOTOS specification of the new design.

It is intended, in the future development of FOOD, that the OO LOTOS designs are hidden beneath some high-level object oriented design interface. The case study, however, required direct manipulation of the OO LOTOS code. At certain key stages in this section, LOTOS code fragments are presented. When appropriate, diagramatic representations of the design (and design components) are given.

The underlying `BankingNetwork` functionality is contained within the ACT ONE implementation of the OO ACT ONE requirements model. This ACT ONE code is not reviewed as part of the thesis. In some instances, new classes of behaviour are required in the design: these classes are coded in OO ACT ONE and then translated to ACT ONE for use in the design.

The case study does not make use of all the CPTs, which are defined in chapter 5. Design is targetted towards a non-concurrent implementation (in Eiffel). As such, the transformations concerned with concurrency and distribution are not illustrated by the case study. However, there is reason to believe that the `BankingNetwork` model could be designed towards a concurrent implementation. An investigation of such a development is beyond the scope of this thesis.

It is not possible to examine, within this thesis, all aspects of the design of the `BankingNetwork` system. The main design steps are reviewed in sections 7.3.1 to 7.3.6. A review of the design process is given in section 7.3.7.

7.3.1 From Analysis to Design: Choosing the Communication Model

Eiffel is the target implementation language and so, for reasons given in chapter 5, the first design decision is to chose the remote procedure call (RPC) model of communication for the `BankingNetwork` system. The initial OO LOTOS design of the `BankingNetwork` is given below in the `PBankingNetwork1` process definition[10].

```
process PBankingNetwork1[newAcc,delAcc,addNode,addLink,switch,accRequest]
(SBankingNetwork: BankingNetwork): noexit:= hide returned, arrived, timeout in
(* EXTERNAL INTERFACE *)
(newAcc?NodeID1:NodeID?Acc1:Acc;
PBankingNetwork[...](.(newAcc( SBankingNetwork, NodeID1, Acc1)))
)[] ...[]
(* INTERNAL TRANSITIONS *)
(returned?MessageID1:MessageID?;
returned!AccTransResultResult(returned(SBankingNetwork, MessageID));
PBankingNetwork[...](.(newAcc( SBankingNetwork, NodeID1, Acc1)))
)[] ...endproc (* PBankingNetwork1 *)
```

This initial design facilitates a review of syntactic conventions in the OO LOTOS designs:

- *Class*es from the requirements model are defined as P*Class* processes in the LOTOS designs. The version of the class definition is identifed by a numeral at the end of the process identifier. As design progresses we verify that correctness is preserved from one version of the process definition to the next.
- The ACT ONE sorts, corresponding to classes in the requirements model, retain the class names used in the analysis. Class variables are represented by the class name preceded by an `S` (for state).

[10] The OO LOTOS process definitions that follow are not given in standard LOTOS syntax. For the sake of brevity, lists of gates, choices and operation parameters are often presented as `listElement1, ...listElement2`, when the context in which this syntax is used makes the identification of the complete list immediate and unambiguous.

- The external attributes of the requirements model classes have two correspondences in the LOTOS designs: as gate names in the process algebra and operation names in the ADT part.
- The ACT ONE operations `.(req(object, ...))` and *Class*`Result(req(object, ...))` are used, respectively, to define the new state of an `object` after servicing a `req`, and the result, if any, returned from servicing a `req`.

7.3.2 Decomposition of the Banking Network System

The initial design is a more concrete implementation of the requirements model since it specifies *how* a `BankingNetwork` communicates with its environment through its external interface. The OO ACT ONE requirements are contained within the ADT part of the specification, and used directly in specifying *what* behaviour is offered at the interface. The initial design does not state *how* the `BankingNetwork` offers this behaviour: it is the designers who must specify *how* this behaviour is to be implemented.

A first step in the design process is to transfer the structure in the requirements into structure in the design: this structure is then amenable to manipulation. The `BankingNetwork` has a static structure and so we apply the static expansion CPT *StExp* to the `PBankingNetwork1` process definition. The second `BankingNetwork` design, resulting from the application of *StExp*, is defined as process `PBankingNetwork2`, below.

```
process PBankingNetwork2[newAcc,delAcc,addNode,addLink,switch,accRequest]
(SBankingNetwork: BankingNetwork): noexit:=
hide returned, arrived, timeout, Transactions1add, Transactions1remove, AccDB1service,
AccDB1delAcc, AccDB1newAcc, AccDB1statement, AccDB1available, Network1switch,
Network1addNode, Network1isNode, Network1getNode, Network1addLink, Network1areConnected in
BankingNetwork2Control[newAcc, ..., Network1areConnected]
|[ Transactions1add, ..., Network1areConnected]|
( PTransactions[Transactions1add, Transactions1remove] (par1(SBankingNetwork)) |||
PAccDB[AccDB1service, ..., AccDB1available] (par2(SBankingNetwork)) |||
PNetwork[ Network1switch, ..., Network1areConnected] (par3(SBankingNetwork))
)endproc (* PBankingNetwork2 *)
```

The `BankingNetwork2` component processes `PAccDB`, `PTransactions` and `PNetwork` are generated as RPC-model designs from their OO ACT ONE specifications. The `BankingNetwork2Control` process co-ordinates the way in which these components are used to provide the external behaviour of `BankingNetwork2`. It is defined below.

The second version of the design is represented in the diagram in figure 7.7, which shows quite effectively the result of the static expansion. Henceforth, similar diagrams are used to illustrate the process structure in the OO LOTOS designs. LOTOS text is included only where necessary.

7.3.3 Decomposition of the Network Component Process

The next design step is the static expansion of the `PNetwork1` process. The result of applying *StExp* to `PNetwork1` is illustrated in figure 7.8. This expansion is necessary because we intend to change the

```
process BankingNetwork2Control[ newAcc,..., Network1areConnected]:  noexit:=
(newAcc?NodeID1:NodeID?Acc1:Acc;
Network1getNode!NodeID1?Node1:Node;
( [and(BoolResult(isControl(Node1), BoolResult(isOn(Node1)))] ->
AccDB1newAcc!Acc1; BankingNetwork2Control[...] )
[](
[not(and(BoolResult(isControl(Node1),BoolResult(isOn(Node1))))] -> BankingNetwork2Control[...]
))[]...endproc (* BankingNetwork2Control *)
```

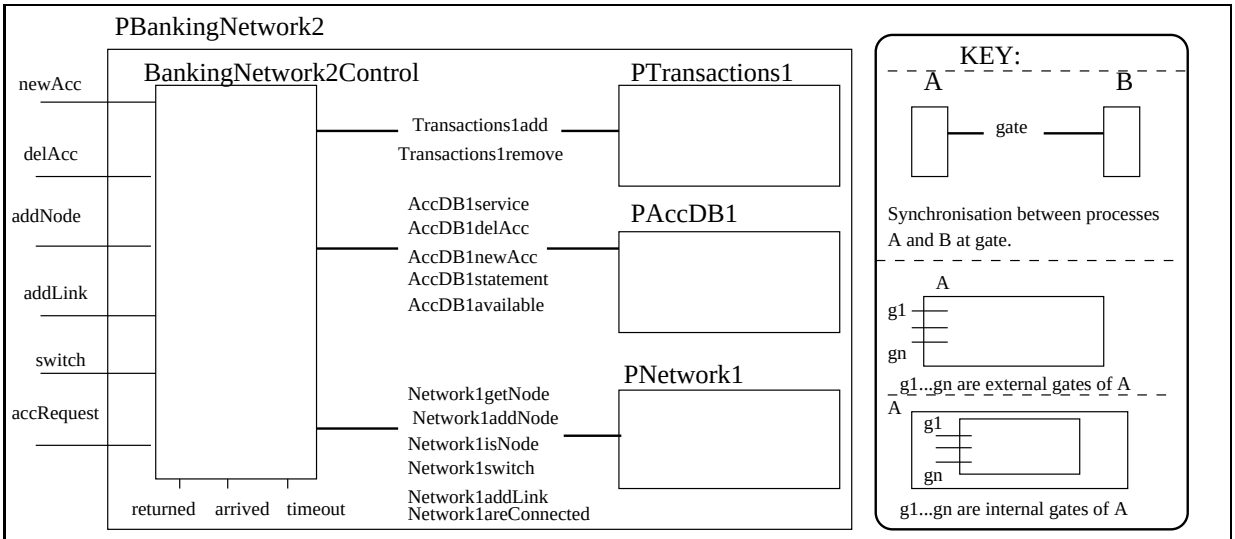


Figure 7.7: BankingNetwork Design Diagram: Stage 2

internal structure of the `Network`.

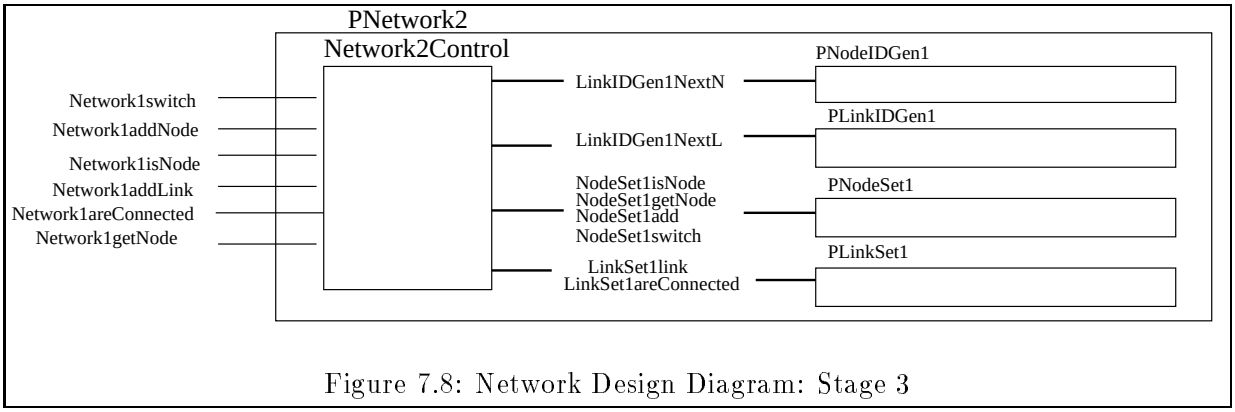


Figure 7.8: Network Design Diagram: Stage 3

7.3.4 Restructuring the Network Component Process

The purpose of the restructuring is to change the way in which `Network` toplogy is defined. Rather than having a distinct set of `Links`, all the `LinkSet` elements are to be distributed between the `Nodes` in the `NodeSet`. This is achieved in two steps:

- i) Compose the `NodeSet` and the `LinkSet` in the `Network`, using the *Comp* CPT.
- ii) Integrate the two recursive structures in the ADT parts of the `NodeSet` and `LinkSet` into one recursive structure of `LinkedNode`s in a `LinkedNodeSet`.

7.3.4.1 Composing NodeSet and LinkSet

The next version of the `Network` design is defined as a composition of the `NodeSet` and `LinkSet` components: `PNetwork3 =` *Comp*`(PNetwork2, {{1}, {2}, {3,4}})`. This new structure is illustrated in figure 7.9.

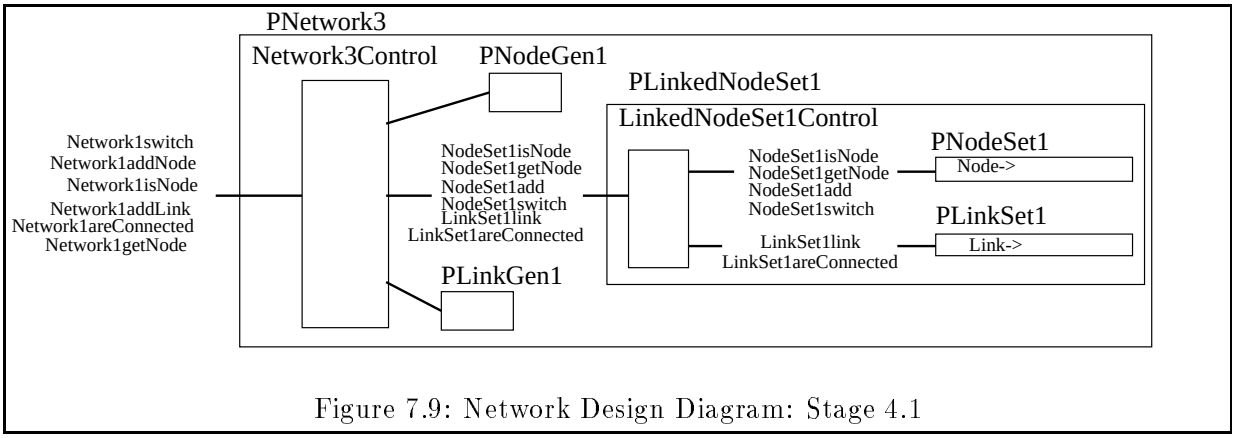


Figure 7.9: Network Design Diagram: Stage 4.1

The initial composition is important because it permits manipulation of the `LinkedNodeSet` specification, independent of the other `Network` components.

7.3.4.2 Merging the NodeSet and the LinkSet

A CPT for the merging of two recursive structures has not been developed as part of this thesis. Consequently, we must examine the transformation in some detail in order to be sure of its correctness.

The idea behind the transformation is very simple: every component of the `LinkSet` should be placed somewhere in the `NodeSet`. The simplest way of achieving this is to give each `Node` in the `NodeSet` an extra component: a `LinkSet`. Then, this extra component can be used to store a subset of the `Network`'s `LinkSet`. We require that every `Link` in the `Network` `LinkSet` is represented in the new set of 'Nodes with Links', named `LinkedNodeSet`. The `LinkedNodeSet` is defined as a recursive structure of `LinkedNode` elements. It is this new set which defines version 2 of the `PLinkedNodeSet` process definition. This new process is illustrated in figure 7.10.

7.3.5 Integrating the Transaction Set in the Network

The mechanism used to merge the `LinkSet` and `NodeSet` is re-applied to merge `Transactions`, a set of `Transaction` components, with the `LinkedNodeSet`. This merging requires an initial restructuring of the `BankingNetwork`, as illustrated in figure 7.11.

The merging requires a new `Transactions` component to be added to the `LinkedNode` structure to define a new class, namely `TransLinkNode`. The `LinkedNodeSet` and `Transactions` components

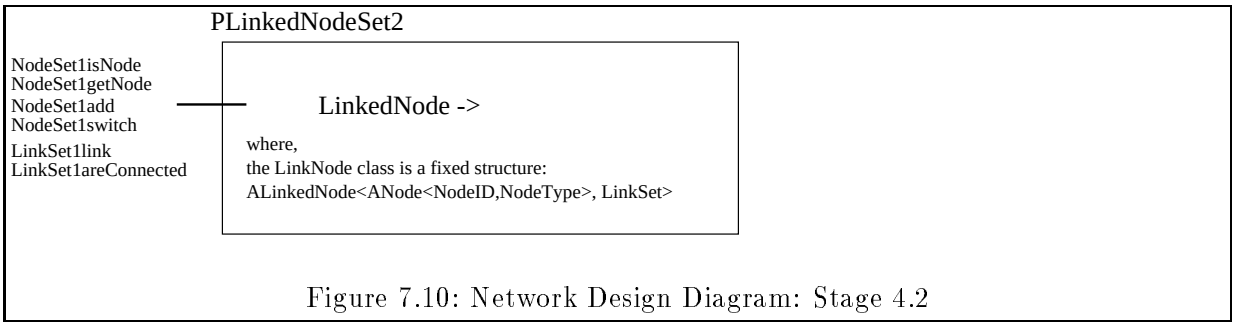


Figure 7.10: Network Design Diagram: Stage 4.2

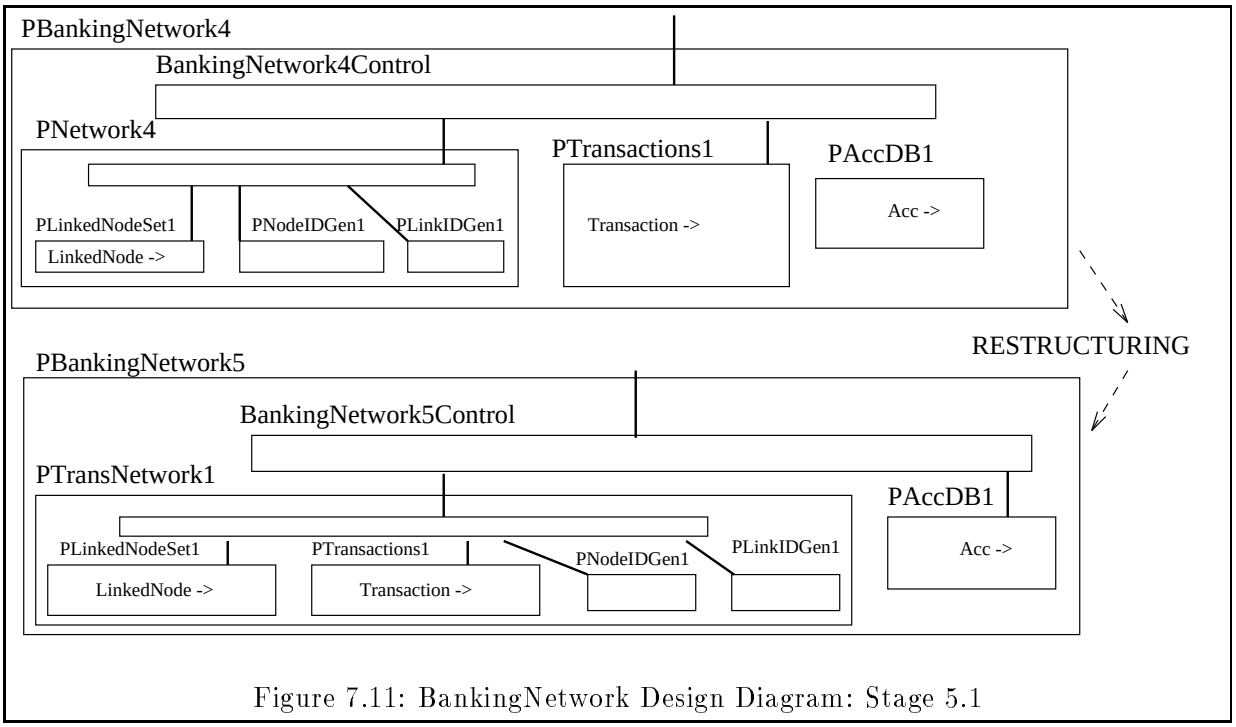


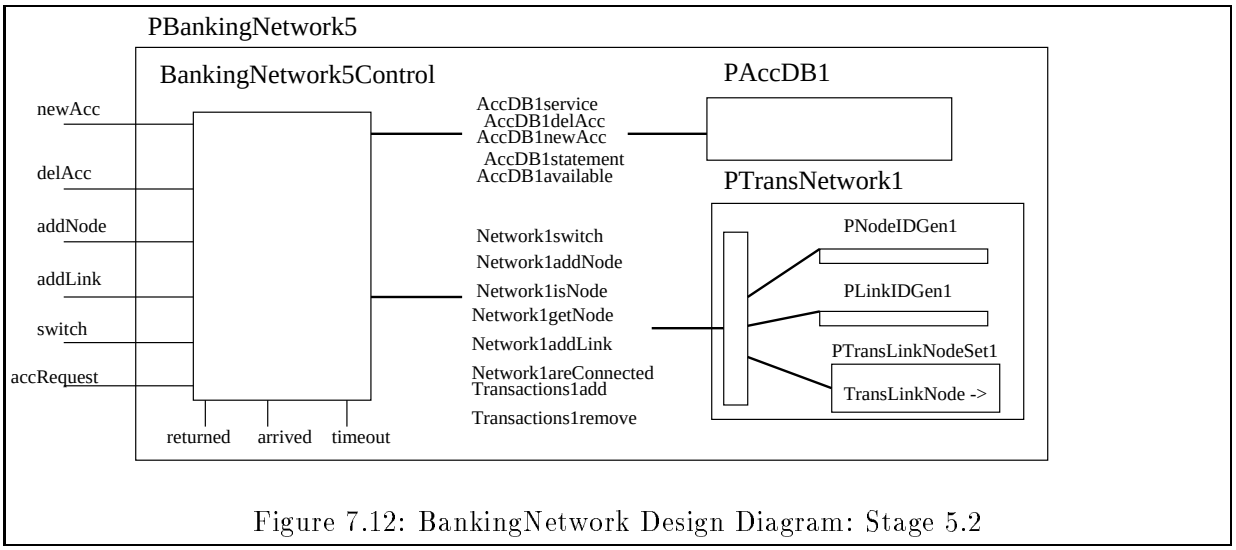
Figure 7.11: BankingNetwork Design Diagram: Stage 5.1

are merged into one component (a `TransLinkNodeSet`) which is defined as a recursive structure of `TransLinkNode` elements. Every `Transaction` in the `BankingNetwork` is recorded in one, and only one, of the `TransLinkNode`s (as a member of its `Transactions` component). The `BankingNetwork` design, after the merging of `Transactions` and the `Network` components, is illustrated in figure 7.12.

7.3.6 An Explicit Routing Mechanism: Removing Nondeterminism

The `BankingNetwork` design has been transformed with the intention of explicitly specifying an internal mechanism for the routing of transaction messages. The `TransLinkNodeSet` class is ready for modelling the routing of messages between nodes. This is done in two stages:

- i) Modelling the movement of messages along links.
- ii) Designing a routing mechanism as a means for nodes to determine on which link outgoing messages are to be sent.



7.3.6.1 Modelling the internal movement of messages

A new internal transition `move<NodeID, NodeID>` is defined to model the movement of a message transaction from the first node to the second (as specified by the `NodeID` parameters). The `TransLinkNode` identified by the first `NodeID` must decide which of its currently held transactions is sent across the link to the node specified by the second `NodeID` parameter. This is determined by the routing mechanism. When a message is recieved by a `TransLinkNode` then one of four things occurs:

- i) The message is a service request which can be serviced by the receiving (control) node. As a result, the `AccDB` is updated appropriately and the message is removed from the network. This models the internal `arrived` transaction.
- ii) The message is an enquiry which can be serviced by the receiving (control) node. As a result, the `AccDB` is accessed to obtain the appropriate reply to the enquiry, the original message is removed from the network and a new message (carrying the reply) is added to the network to be routed back to the node which originated the enquiry.
- iii) The message is a reply which has arrived back at the node which originated the enquiry. The information is given to the receiving node and the message is removed from the network. This models the `returned` transaction.
- iv) The message cannot be serviced by the receiving node so it is forwarded for routing to another node.

The internal `timeout` transformation is defined to remove the specified message from withinwhichever `TransLinkNode` it is stored.

7.3.6.2 Designing A Routing Mechanism

The design needs to be extended to provide each `TransNode` with a routing mechanism. [55] examines a number of different network routing mechanisms: their specification in LOTOS and resulting

implementation in C++. In the `BankingNetwork` case study, one of the simplest routing mechanisms was chosen for implementation, namely the *Hot Potato* mechanism, see [102].

The *Hot Potato* routing algorithm is developed on the philosophy that it is always best to get rid of an incoming message (that needs routing) as quickly as possible. To model this, we define every `TransLinkNode` to have an additional component, namely a set of transition queues (one queue for each outgoing link in the node). Consequently, the `TransLinkNodeSet` of `TransLinkNode` components is replaced by a `RoutingNodeSet` of `RoutingNode` components. The movement of messages between nodes is simply achieved by popping a transaction off the queue, identified by a `LinkID`.

The specification of *Hot Potato* routing behaviour (and a number of extensions and refinements) is given in [55]. This behaviour is incorporated in the `BankingNetwork` in a straightforward manner (many of the classes are directly re-used). The final design of the `BankingNetwork` is illustrated in figure 7.13. It is this design which is put forward for implementation in Eiffel.

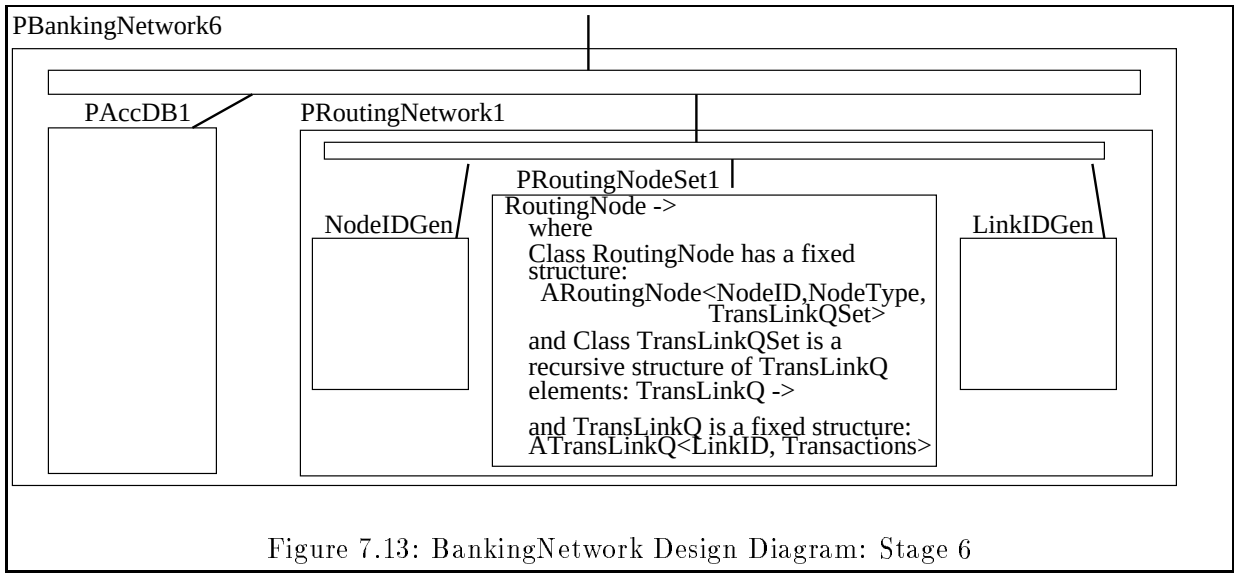


Figure 7.13: BankingNetwork Design Diagram: Stage 6

7.3.7 A Review of the Design Process

7.3.7.1 Limitations of the RPC-model of communication

Designing towards a non-concurrent implementation is much simpler than designing towards a concurrent implementation. With an RPC-model of communication, the design decisions are primarily concerned with:

- Composition and decomposition of data structure.
- Removing nondeterminism.

Unfortunately, the CPTs which address concurrency and distribution design decisions are not relevant in a non-concurrent model. It is these issues which the make the transformation to full LOTOS from OO ACT ONE a vital part of FOOD. This is not fully illustrated by the case study.

7.3.7.2 Towards A Concurrent Implementation

The case study addresses a problem which is inherently concurrent: the `BankingNetwork` is modelling a system of distributed `Node` processes. This concurrency is designed out of the system by introducing internal transformations for modelling the routing of messages between nodes. Given a concurrent target implementation language, the FOOD approach can target the design using CPTs, defined in chapter 5, which were not utilised in the case study. An area of future work is the investigation of the effectiveness of these other CPTs. A concurrent implementation of the `BankingNetwork` provides a suitable problem for such an investigation.

7.3.7.3 The Need For Tool Support

This thesis is principally concerned with mathematical models, i.e. conceptual tools. The CPT-driven design is particularly rich in this respect. Conceptual tools must be supported by development environments. The LOTOS tools (both the SEDOS tools and LITE tools were used) provided a limited support during the design stages of FOOD. Unfortunately, these tools do not support object oriented design: they support the development of LOTOS specifications. It is clear that the design stages of FOOD need more suitable tool support.

7.3.7.4 The Need For More CPTs

The case study illustrates the importance of CPTs in design. It is vital that a comprehensive set of object oriented design CPTs are developed. This will improve the quality of design and the productivity of designers. In short, CPTs allow designers to decide *what* should be done, rather than *how* it can be done.

This thesis introduces only a small number of CPTs. These are used to show that a CPT-driven approach to object oriented design is possible. However, the case study shows that many more CPTs are needed: for example, the merging of two recursive structures needs to be treated formally. It is hoped that CPTs become as widely re-used as classes of behaviour.

7.4 The Eiffel Implementation

It is not effective to examine the Eiffel code for the `BankingNetwork` (this code is available on request). Rather, an overview is given of the implementation process.

7.4.1 The Role of the Final Object Oriented LOTOS Design

After the development of the requirements model, and its manipulation in the design, the Eiffel implementation went as outlined in section 6.4. The final design had a direct influence on implementation since there was a correspondence between:

- Classes in the design and classes in the implementation.
- Class hierarchies in the design and class hierarchies in the implementation.

- Composition structure in the design and composition structure in the implementation.

7.4.2 Re-Use in the Implementation

The extensive library of Eiffel classes meant that many of the design classes were already implemented:

- The `Sum` class (in the account database) was implemented as a real number in Eiffel.
- The `TransLinkQ` class (in the network) was implemented as an instance of the generic queue class in Eiffel.
- The different identifier classes were all implemented as Eiffel integers.
- The recursive structures were implemented as Eiffel linked lists.

7.4.3 Implementing Exceptions

The exceptions in the requirements model were not designed out. Eiffel, as the target implementation language, provides an exception handling construct. Consequently, design did not have to address this behaviour. The handling of exceptions was straightforward: warning messages were returned to the user interface and the state of the system before the exception was returned to.

7.4.4 Implementing A User Interface

Perhaps surprisingly, the coding of the user-interface required more time than the coding of the functionality being provided at the interface. Initially, a graphical user interface to the `BankingNetwork` was developed in Eiffel. However, after struggling with many of the problems in the Eiffel class library, a simple textual interface was coded instead.

7.5 A Review of the Case Study

7.5.1 Development Statistics

An important aspect of the case study is the way in which the development effort was distributed between the three main stages of FOOD: requirements capture, design and implementation. Although it is difficult to be precise about development costs, the following statistics may be useful:

- **Requirements Capture**
  Analysis and requirements capture required approximately one man month of work. It resulted in 1200 lines of OO ACT ONE code, which was translated into 6500 lines of ACT ONE. The requirements model contained 25 classes of behaviour, with an average of 6 external attributes per class. Over one half of the classes had a fixed structure.
- **Design**
  Design required approximately three man weeks of work. It resulted in 1400 lines of LOTOS process algebra together with 7400 lines of ACT ONE. Design introduced 8 new classes of behaviour.

- **Implementation**
  Implementation required approximately one man month of work (of which more than half was spent developing a user-interface). The Eiffel code (ignoring the user interface) was approximately 800 lines. The reduction in code was a consequence of three things:
  - The OO LOTOS design had to explicitly define the communication model, whilst this is explicit in the Eiffel semantics.
  - The reference semantics of Eiffel reduced the amount of code needed to define the routing of messages between nodes.
  - The extensive use of pre-coded components.

The above statistics strengthen our claim that FOOD places much more emphasis on the earlier stages of development. This claim can be properly verified only if FOOD is used in a wide range of software engineering projects.

7.5.2 The Effectiveness of FOOD

The case study shows the effectiveness of FOOD arising out of the combination of formal and object oriented methods:

- **Formality**
  A formal approach improved understanding, removed potential errors earlier in the development and made design decisions explicit.
- **Object Orientedness**
  An object oriented approach provided a conceptual integrity, facilitated re-use at all stages of development and supported opportunistic development.

7.5.3 Extensions to the Behaviour

To illustrate the way in which FOOD supports extensions to systems that have already been developed, two extensions were considered: changing the routing mechanisms and providing more banking services.

7.5.3.1 Changing Routing Mechanisms

The `BankingNetwork` incorporated a very simple routing mechanism: the hot potato algorithm. Two other, more complex routing mechanisms replaced the hot potato method. These new mechanisms were integrated into the system by first making changes to the design and then updating the Eiffel code:

- **A Flooding Mechanism**
  In this model, incoming messages are sent on all outgoing links and a constraint was introduced so that messages were moved only a limited number of times. This required approximately one man week of work.

- **A Backward Learning Mechanism**
  Each node was extended to incorporate a store of the shortest paths to other nodes. This store is updated when an incoming message has come by a route which was shorter than that stored. Consequently, every message must carry additonal routing information. This change required approximately three man weeks of work. The additional routing behaviour was very complex and so, as a preliminary to design, OO ACT ONE was used to construct a routing requirements model. This improved our understanding of the behaviour required and the design involved making decisions as to how this behaviour was to be implemented.

7.5.3.2 Providing Additional Banking Functionality

The facilities provided for each bank account were extended to include a mechanism for transferring money from one account to another. This extension was carried through by first making changes to the requirements model and then proceeding with design and implementation. The updated bank accounts were defined as subclass (extensions) of the original classes and the design structure needed minimal changes. This whole process took less than one man day.

7.5.3.3 Lessons Learned From Extensions

Four important lessons were learned from the case study:

- Re-use through composition is much more common than re-use through inheritance.
- When building a system it is important to keep potential extensions in mind.
- Meaningful generalisation should be applied when possible: the extra development time is rewarded with benefits in later projects.
- Components that can be extended within one application area are not necessarily re-usable in other application areas: for example, it is unlikely that the routing mechanisms can be usefully re-used in other problem domains.

7.5.4 The Importance of Structure Throughout Development

The case study places emphasis on structure at all stages of development. The advantages of maintaining structure from specification to implementation are as follows:

- Traceability, in the sense of a design audit, is improved. Testing the implementation can be done in a constructive fashion
- Extending or changing a system can be achieved in a controlled fashion. With an object oriented approach, modifications are often localised (resulting in changes to only a few classes in the system).
- Structure provides the framework upon which mutual understanding, between different members of the software development team, is based.

Chapter 8

Conclusions

This chapter reviews the objectives of the thesis, shows how the work presented in this thesis meets these objectives and makes suggestions for future work.

8.1 Review of Thesis Objectives

The main objective of the thesis is to show that combining object oriented and formal methods is a practical and effective way of improving the software development process. To meet this objective the thesis addresses five separate goals:

- To record an understanding of software devlopment and, using this understanding, to formulate an *ideal* software development environment.
- To show that a formal object oriented development method is a step towards achieving such an *ideal*.
- To remove the ambiguity and informal nature of object oriented terminology by developing an object oriented semantics.
- To construct a formal object oriented development (FOOD) framework based on these object oriented semantics.
- To illustrate the effectiveness of FOOD by applying it in the development of a non-trivial software system.

8.2 Meeting Objectives: The Contributions of the Thesis

Chapter one establishes the complementary nature of object oriented and formal methods within the domain of software engineering. The thesis is developed on the premise that *correctness* is the most important property of software, and argues that *software engineering* must be based on mathematical models. Object oriented methods are presented as providing a practical solution to the synthesis and analysis of mathematical models of computer systems, in general, and software in particular. LOTOS is proposed as a good language for implementing object oriented semantics at all stages of

software development. The ADT part of LOTOS is shown to be suitable for implementing abstract requirements models, whilst full LOTOS is put forward as an ideal language for incorporating these abstract requirements in a more concrete design model.

The integration of a process algebra and ADT, within LOTOS, is the main reason for its use within this thesis. It provides a smooth transition from requirements to design by facilitating the implementation of a class at various levels of abstraction within the same semantic framework: as an abstract data type in the requirements models to a process in the design models. This novel approach to using LOTOS is a major contribution of this thesis.

Chapter two introduces object oriented methods by first considering object oriented analysis and its relation to other analysis methods. It motivates the development of a formal approach to analysis and requirements capture. The chapter provides a rationale for the success of object oriented development methods and, based on this rationale, proposes a set of object oriented models which can be used throughout software development. A contribution of the thesis is a recognition of the importance of the way in which these models are co-ordinated. Further, the thesis identifies the importance of establishing a semantics for object oriented terminology which is consistent throughout the development process. As a system moves from the abstract to the concrete it is fundamental to successful development that leaps between different semantic frameworks are curtailed. The thesis proposes a mechanism for constructing design models in which the requirements are still present. In chapter 2, abstract data types are shown to provide a level of abstraction suitable for the specification of object oriented requirements. However, as types are more general than classes, we argue that it is necessary to develop a more abstract view of objects and classes.

Chapter three includes one of the major contributions of the thesis: an abstract object oriented semantics based on the modelling of state transitions. We argue that the state transition view is ideal for communicating object oriented requirements. An *object-labelled state transition system* (O-LSTS) semantics is developed, and the object oriented notions of classification, subclassing, composition, configuration and interaction are formally defined. The O-LSTS semantics are then used in the definition of an object oriented analysis language (OO ACT ONE), which, as its name suggests, is syntactically similar to ACT ONE (but with a distinct object oriented 'sugaring'). An important contribution of this thesis is the formulation of two different types of subclassing, namely extension and specialisation, and the provision of language mechanisms for defining class hierarchies based on these relationships. The thesis argues that these two mechanisms are sufficient, during analysis, for the definition of all subclassing relationships. In conclusion, chapter three identifies the need for the requirements models to be executed: *customer validation* is argued to be dependent on the ability to step through a dynamic execution of the requirements models. A translation to ACT ONE provides a means of stepping through the dynamic behaviour of an OO ACT ONE specification.

Chapter 4 considers how the formal object oriented models can be successfully used in the initial stages of software development. In particular, customer-analyst communication is identified as the most important aspect of requirements capture. This chapter examines how the analyst, using the formal object oriented models, can achieve a mutual understanding of the problem domain with the customer. The difficult question of how the analyst can and should alter the way in which a cus-

tomer conceptualises their needs is considered. The chapter concludes by re-iterating the importance of constructive specification in an object oriented development strategy. The thesis argues that the structure of the problem domain should be recorded in the requirements model: it improves understanding and acts as a framework upon which design can begin. Designers must be given the option of reproducing the problem domain structure in their designs. Without this option, object oriented design can be very difficult.

Chapter five considers the role of design within software engineering. The thesis shows that *constructive* design, based on the application of *correctness preserving transformations*, is the most practical solution to the problem of ensuring that design meets requirements. The way in which the ACT ONE executable model of requirements is incorporated in the full LOTOS designs is an original and effective means of going from analysis to design. We argue that a process algebra is a useful conceptual tool for the specification of communication models, which is fundamental to object oriented design. Chapter five shows that there are a potentially very large number ways of using LOTOS to model objects and classes. These provide different object oriented semantics with respect to the way in which models communicate and interact. The thesis argues that it is the role of designers to choose a model which is best suited to their target implementation environment. A number of different, though equally valid, object oriented communication models are put forward. A major contribution of the thesis is the formulation of a means of generating different designs from the same requirements, all of which maintain *correctness*.

Chapter five also contributes a small number of CPTs for the manipulation of composition structure within object oriented designs. The ability to restructure the composition of classes is shown to be fundamental in object oriented design: targetting design towards classes which have already been implemented is dependent on the ability of designers to restructure their designs whilst maintaining correctness. The transfer of structure in the ADT part of a LOTOS design to structure in the process algebra part is shown to be an important step in the movement from abstract to concrete. Chapter five emphasises the importance of designers understanding the facilities provided by their target implementation environment. The role of designers is defined as restructuring the requirements from being *customer oriented* to being *implementation oriented*. As with requirements capture, we emphasise that the design models provide only a framework for the development of a design method.

Chapter six considers the implementation of OO LOTOS designs. A major contribution of this thesis is the formulation of general strategies for implementing the formal object oriented designs. The importance of having a fundamental understanding of implementation language semantics is emphasised. The thesis shows, in some detail, how appropriately targetted OO LOTOS designs can be implemented in Eiffel. The thesis also illustrates how FOOD is well suited to the development of concurrent software.

Chapters two to six provide a framework of models and techniques for using these models in the development of software. Chapter seven argues that a software development method must evolve from the use of models rather than being an immediate consequence of their formulation. It also argues that practical use of models and methods is the only true means of evaluating their effectiveness. Consequently, as part of this thesis, chapter seven reports on a case study in which FOOD was used

to develop a non-trivial software system. This contributes to the thesis by placing the theory in a more practical domain.

As a whole, the thesis identifies the problems inherent in software development, shows how a formal object oriented method can overcome many of these problems and provides a framework upon which such a method can evolve. In short, FOOD shows that, although much more work needs to be done, it has the potential to result in an efficient means of producing software which fulfils its requirements.

8.3 Future Work

FOOD is a framework of models and methods which provides the basis upon which an *ideal* software development environment can be constructed. To get closer to this *ideal* much more work needs to be done:

- **Analysis and requirements capture**
  A natural step forward is to provide a direct implementation of OO ACT ONE specifications, rather than translating them to ACT ONE. Such an implementation could be incorporated in a comprehensive set of analysis and synthesis tools. As the customer is central to analysis and requirements capture, it is important to further investigate the process of customer-analyst interaction within a formal object oriented framework. The diagramatic representations of the object oriented requirements must be made an integral part of the development environment. It is important that a means of letting the customer directly interact with the requirements models is developed. This interaction can be used in the construction and validation of requirements models.
- **Design**
  In the thesis, LOTOS is used to provide our object oriented design semantics. This is fine in a prototype development environment, but it is important that a cleaner object oriented design language is developed. This should be a superset of OO ACT ONE which incorporates semantics for processes, inter-process communication, nondeterminism and concurrency. Then the CPTs must be translated for designing in this new language. It is vital that the set of CPTs is widely expanded. This can only be done if the process of object oriented design is more thoroughly analysed: the most common design decisions must be identified and CPTs defined to model these decisions. Further, a means for designers to develop, record and re-use CPTs needs to be formulated. This should be incorporated as part of a set of tools for the analysis, synthesis and manipulation of designs.
- **Implementation**
  The implementation stage of FOOD is perhaps the weak point in the whole development strategy. Targetting designs to an informal semantics is not a suitable final step in a formal development method. A natural means of getting around this problem is to build an implementation language on top of the FOOD semantics. Then, the final implementation step can be given a

formal basis. It is important, for future development, that such a language has a concurrent semantics.

There are other areas of work, applicable to all stages of development, which arise out of the thesis:

- **Re-use**
  FOOD promotes re-use at all stages of development. The consequences of developing for re-use and with re-use need to be addressed. In particular, the heuristics for costing software development using FOOD need to be examined. Further, there needs to be some method for controlling the storing of, and access to, classes of re-usable behaviour at different levels of abstraction.
- **Evolving Method**
  Only through widespread application can FOOD become a software development method. Consequently, it is necessary that FOOD is used in a wide range of case studies, each of which learns from previous development. In this respect, FOOD needs to be extended to incorporate many of the *management* aspects which are common in the most popular software development techniques. Future work must atempt to derive a rational for software development method which can be incorporated in FOOD.

In conclusion, we believe that the development of a practical and effective industrial strength software development method, based on FOOD, is feasible in the future.

Bibliography

[1] P. America. Object oriented programming: a theoretician's introduction. Technical report, Philip's Research Laboratories, Eindhoven, 1988.

[2] Lee Atkinson and Mark Atkinson. *Using C/C++ Special Edition.* QUE Corporation, 1992.

[3] R.L. Baber. *The Spine of Software — Designing Provably Correct Software: Theory and Practice, or: A Mathematical Introduction To The Semantics Of Computer Programs.* John Wiley and Sons, 1987.

[4] Emery Berger. Fp + oop = haskell. Technical report, Department of Computer Science, University of Texas at Austin, 1992.

[5] D. Bjoener and C. B. Jones. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[6] S. Black. Objects and LOTOS. Technical report, Hewlett-Packard Laboratories, Stoke Gifford, Bristol, 1989.

[7] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, Stefik M., and Zdybel F. Commonloops: Merging Lisp and object-oriented programming. In *ACM SIGPLAN Notices*, pages 17–29, November 1986.

[8] P. Boehm et al. Towards practical verification of LOTOS specifications. Esprit/sedos/n.121, University of Twente, October 1987.

[9] T. Bolognesi. Fundamental results in the verification of observational equivalence: a survey. In H. Rudin and West C.H., editors, *Protocol Specification, Testing and Verification VII.* North-Holland, 1988.

[10] T. Bolognesi and M. Caneve. SQUIGGLES: A tool for the analysis of LOTOS specifications. In K.J.T. Turner, editor, *The 2nd International Conference on Formal Description Techniques (FORTE 89)*, 1989.

[11] T. Bolognesi and F. Lucidi. LOTOS-like process algebras with urgent or timed interactions. In K. Parker and G. Rose, editors, *Formal Description Techniques IV*, pages 249–264. North-Holland, 1992.

[12] G. Booch. *Object Oriented Development.* IEE Software Engineering, February 1986.

[13] G. Booch. *Object oriented design with applications.* Benjamin Cummings, 1991.

[14] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Springer-Verlag, 1991. Lecture Notes in Computing Science, number 562.

[15] E. Brinksma and Scollo G. Formal notions of implementation and conformance in lotos. Mem: INT-86-13, University of Twente, NL, December 1986.

[16] Ed. Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS specifications, their implementation and their tests. In *Sixth International Symposium on Protocol Testing, Specification and Verification*, Montreal, June 1986.

[17] D. Budgen. Introduction to software design. Curriculum module SEI-CM-2.2.1, Carnegie-Mellon University, January 1989.

[18] D. Budgen. *Software Development with Modula-2.* Addison-Wesley, 1989.

[19] S. Budkowski and P. Dembinkski. An introduction to ESTELLE: A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14(1):3–23, 1987.

[20] R. Bulzer and N. Goldman. Principles of good software specification and their implications for specification languages. In *Proc. of Reliable Software*, pages 58–67. Cambridge, Mass., 1979.

[21] H. I. Cannon. Flavours. Technical report, MIT International Laboratories, Cambridge (Mass), 1980.

[22] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[23] R. G. G. Cattell and T. R. Rogers. Combining object-oriented and relational models of data. In *International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[24] Robert Clark. Using LOTOS in the object based development of embedded systems. In *The Unified Computation Laboratory.* The Institute of Mathematics and its Applications (OUP), 1991.

[25] P. Coad and E. Yourdon. *Object oriented analysis.* Prentice-Hall (Yourdon Press), 1990.

[26] P. Coad and E. Yourdon. *Object oriented design.* Prentice-Hall (Yourdon Press), 1990.

[27] L. Constantine. Beyond the madness of methods: System structure methods and converging design. In *Software Development 1989.* Miller-Freeman, 1989.

[28] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceeedings of 19th ACM Symposium on Principles of Programming Systems and Languages*, pages 125–135, 1989.

[29] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Object Oriented Programming Languages Systems and Applications (OOPSLA 89)*, 1989.

[30] W. R. Cook. Proposal for making Eiffel typesafe. *Computer Journal*, 32(4), 1989.

[31] Brad Cox. *Object oriented programming: an evolutionary approach.* Addison-Wesley, 1986.

[32] Dan Craigen, Susan Gerhart, and Ted Ralston. An international survey of industrial applications of formal methods. Nistgcr 93/626, U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Lab., Gaithersburg, MD 20899, 1993.

[33] E. Cusack. Formal object oriented specification of distributed systems. In *Specification and Verification of Concurrent Systems*, University of Stirling, 1988.

[34] E. Cusack. Refinement, conformance and inheritance. In *Open University workshop on the theory and practice of refinement*, 1989.

[35] E. Cusack, S. Rudkin, and C. Smith. An object oriented interpretation of LOTOS. In K.J.T. Turner, editor, *The 2nd International Conference on Formal Description Techniques (FORTE 89)*, 1989.

[36] Geoff Cutts. *Structured system analysis and design method.* Blackwell Scientific Publishers, 1991.

[37] O. Dahl. Object-oriented specification. In P. Wegner and B. Shriver, editors, *Research Directions in Object Oriented Programming.* MIT Press, 1987.

[38] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[39] D. L. Davies and Barber D. L. A. *Computer Networks and their Protocols.* John Wiley, 1979. Section 3.

[40] J.W. de Bakker. *Mathematical Theory of Programming Correctness.* Prentice-Hall, 1980.

[41] T. DeMarco. *Structured analysis and system specification.* Prentice-Hall, 1979.

[42] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: an overview. In J. Bezivin, J. Hullot, P. Cointe, and H. Lieberman, editors, *Proceeedings of the 1987 European Conference on Object Oriented Programming (ECOOP 87)*, pages 151–170. Springer-Verlag, June 1987.

[43] R. DeNicola. Extensional equivalence for transition systems. *Acta Informatica*, 24:211–237, 1987.

[44] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEE Software Engineering*, SE-2:80–86, June 1976.

[45] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall International, 1976.

[46] A. Diller. *An Introduction To Formal Methods.* John Wiley and Sons, 1990.

[47] T. B. Dinesh. *Object-Oriented Programming: Inheritance to Adoption.* PhD thesis, University of Iowa, 1992.

[48] H. Ehrig and Mahr B. *Fundamentals of Algebraic Specification I.* Springer-Verlag, Berlin, 1985. EATCS Monographs on Theoretical Computer Science (6).

[49] R. Fairley. *Software Engineering Concepts.* McGraw Hill, New York, 1985.

[50] D. Freestone, M. Norris, and K. Odam. Towards better system specification. *British Telecom Technology Journal*, 3, July 1986.

[51] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and techniques.* Prentice-Hall, 1979.

[52] R. J. Gautier and P. J. L. Wallis, editors. *Software Re-use with Ada.* Peter Peregrinus (on behalf of IEE), 1990.

[53] J. Paul Gibson. Object oriented analysis and design principles: A proposal for their incorporation into splice. Internal SPLICE document jpg2, 1992.

[54] J. Paul Gibson. Structured analysis and design methods and models: Investigating their relevance to splice. Internal SPLICE document jpg3, 1992.

[55] J.Paul Gibson and D. Freer. Applying LOTOS in an object oriented development strategy. Internal BT Technical Report, Formal Methods Group (RT6222), February 1991.

[56] J.Paul Gibson and Lynch J.A. Applying formal object oriented design principles to Smalltalk-80. *British Telecom Technology Journal*, 3, July 1989.

[57] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[58] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1983.

[59] Joseph Gougen. Reusing and interconnecting software components. *Computer*, 20, February 1986.

[60] Joseph Gougen and David Wolfram. On types and FOOPS. Programming Research Group, Oxford University, Draft Report, 1990.

[61] R. Guillemot, M. Haj-Hussein, and L. Logroppo. Executing large LOTOS specifications. In *Proceedings of Prototyping, Specification, Testing and Verification VIII*. North-Holland, 1991.

[62] Raymonde Guindon. Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies*, 33(3):279–304, 1990.

[63] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. In *Proceeedings of the 10th IEEE Real Time System Symposium*, pages 102–111. Computer Society Press, 1989.

[64] I.J. Hayes and C.B. Jones. Specifications are not necessarily executable. Technical report, Department of Computing Science, University of Queensland, Australia, December 1989.

[65] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[66] Sommerville I. *Software Engineering*. Addison-Wesley, 3rd edition, 1989.

[67] IEE. *Special Collection On Requirements Analysis*. IEE Transactions on Software Engineering, 1977.

[68] ISO. LOTOS — a formal description technique based on the temporal ordering of observed behaviour. Technical report, International Organisation for Standardisation IS 8807, 1988.

[69] M.A. Jackson. *System Development*. Prentice-Hall, 1983.

[70] R. E. Johnstone and B. Foote. Designing re-usable classes. *Journal of Object Oriented Programming JOOP*, pages 22–35, June 1988.

[71] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International, 1980.

[72] C. B. Jones. *Sytematic Software Development Using VDM*. Prentice-Hall International, 1986.

[73] S. E. Keene. *Object-oriented programming in Common Lisp: a programmer's guide to CLOS*. Addison-Wesley, 1989.

[74] W. Kim and F. Lochovsky. *Object Oriented Concepts, Databases, and Applications*. ACM Press, New York, 1988.

[75] M. Lai and E. Cusack. Object oriented specification in LOTOS and Z or, my cat really is object oriented. In de Bakker, J. W. et al., editors, *Proc. Foundations of Object Oriented Languages*, pages 179–202. Springer Verlag, 1991. Lecture Notes in Computer Science, Number 489.

[76] Jintae Lee and Kum-Yew Lai. What's in design rationale? *Human-Computer Interaction*, 6(3&4):251–280, 1991.

[77] K. Lee, S. Rudkin, and K. Chon. Specification of a sieve object in objective LOTOS. Technical report, British Telecom Research Laboratories (Formal Methods Group), St. Vincent House, Ipswich, 1990.

[78] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[79] A. Malhotra, J. C. Thomas, J. M. Carroll, and L. A. Miller. Cognitive processes in design. *International Journal of Man-Machine Studies*, 12(2):119–140, 1980.

[80] D.A. Marca and C. L. McGowan. *SADT: Structured Analysis and Design Technique*. McGraw-Hill, 1988.

[81] T. Mayr. Specification of object oriented systems in LOTOS. In *The 1st International Conference on Formal Description Techniques (FORTE 88)*, 1988.

[82] B. Meyer. Genericity versus inheritance. In *Object Oriented Programming Languages Systems and Applications (OOPSLA 86) As ACM SIGPLAN 21*, November 1986.

[83] B. Meyer. Re-usability: the case for object oriented design. *IEE Software Engineering*, March 1987.

[84] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[85] B. Meyer. You can write, but can you type? *Journal of Object Oriented Programming JOOP*, March 1989.

[86] B. Meyer. *Eiffel: The Language*. Prentice Hall International Ltd., 1992.

[87] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[88] D. A. Moon. Object-oriented programming with Flavours. In *ACM SIGPLAN Notices*, pages 1–8, November 1986.

[89] Ana Moreira and Robert Clark. Object oriented analysis and its relation to object oriented design. Technical report, Stirling University Computing Science and Mathematics Department, 1992.

[90] G. J. Myers. *Composite Structure Design*. Van Nostrand Reinhold (NY), 1978.

[91] Kristen Nygaard and Ole-Johan Dahl. Simula 67. In Richard W. Wenenlblat, editor, *History of Programming Languages*. Wenelblat, 1981.

[92] K. Ohmaki, K. Futatsugi, and K. Takahashi. A basic LOTOS simulator in OBJ. Computer Language Section, Computer Science Division, Electrotechnical Laboratory, 1-1-4 Umezono, Japan, Draft Report, 1990.

[93] K. Orr. *Structured Systems Development*. Yourdon Press, 1977.

[94] M. Page-Jones. *The Practical Guide to Structured System Design*. Yourdon Press, 1988.

[95] M. Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming*. PhD thesis, University of Geneva, 1992.

[96] Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach To Software Development*. Springer-Verlag, 1990.

[97] K. Raymond, P. Stocks, and D. Carrington. Specifying ODP systems in z. Technical report, University of Queensland, March 1990.

[98] D.W. Reynolds. Software reusability and its implications. Research and technology report rt32/033/89, British Telecom, 1989.

[99] D. T. Ross. Structured analysis (SA): A language for communicating ideas. In *IEE Transactions on Software Engineering*. IEE, 1977.

[100] S. Rudkin. Inheritance in LOTOS. In K. Parker and G. Rose, editors, *Formal Description Techniques IV*. North-Holland, 1991.

[101] James Rumbaugh et al. *Object oriented Modeling and Design*. Prentice-Hall, 1991.

[102] M. Schwartz and T. E. Stern. Routing techniques used in communicating computer networks. *IEE Transactions on Communications*, 28(4), April 1987.

[103] A. Snyder. Common objects: an overview. *ACM SigPlan Notices*, October 1986.

[104] L. A. Stein. Delegation is inheritance. In *Object Oriented Programming Languages Systems and Applications (OOPSLA 87)*, 1987.

[105] R. G. Stone and D. J. Cooke. *Program Construction.* Cambridge Computing Science Texts 22. Cambridge University Press, 1987.

[106] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986.

[107] William Swartout and Robert Balzer. On the inevitable intertwining of specification and implementation. *Comm. ACM*, 25(7), July 1982.

[108] D. Taenzer, M. Grant, and S. Poder. Problems in object-oriented software re-use. In S. Cook, editor, *Proceeedings of the 1989 European Conference on Object Oriented Programming (ECOOP 89)*, pages 25–39. Cambridge University Press (on behalf of British Computer Society), 1989.

[109] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, 1981.

[110] D. Thomas. In search of an object oriented development process. *Journal of Object Oriented Programming JOOP*, May 1989.

[111] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag, September 1985.

[112] Kenneth J. Turner. A LOTOS-based development strategy. Technical report, Department of Computing Science, University of Stirling, 1989.

[113] Kenneth J. Turner. The formal specification language LOTOS: A course for users. Technical report, Department of Computing Science, University of Stirling, 1990.

[114] K.J. Turner. SPLICE I: Specification using LOTOS for an interactive customer environment — phase 1. University of Stirling SPLICE Internal Technical Document, 1992.

[115] K.J.T. Turner. *Using FDTS: An Introduction To ESTELLE, LOTOS and SDL.* John Wiley and Sons, 1993.

[116] Turski and Malibaum. *The Specification of Computer Programs.* Addison Wesley, 1987.

[117] van Eijk, Vissers, and Diaz. *The Formal Description Technique LOTOS.* North-Holland, Amsterdam, 1989.

[118] W.H.P. van Hulzen. Object oriented specification style in LOTOS. Lo/wp1/t1.1/rnl/n00002, LOTOSPHERE, 1989.

[119] W. Visser. More or less following a plan during design: opportunistic deviations in specification. *International Journal of Man-Machine Studies*, 33(3):247–278, 1990.

[120] C. Vissers et al. On the use of specification styles in the design of distributed systems. Technical report, University of Twente, Fac. Informatics, 7500 AE Enschede, NL, 1989.

[121] R. Waddell, Gordon. An analysis of object oriented development using eiffel. Technical report, Stirling University, Department of Computing and Mathematics, Honours Project, 1992.

[122] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceeedings of 16th ACM Symposium on Principles of Programming Systems and Languages*, pages 60–76, 1989.

[123] D. Walker. π calculus semantics of object-oriented programming languages. Technical Report ECS-LFCS-90-122, Computer Science Department, Edinburgh University, Laboratory for Foundations of Computer Science, October 1990.

[124] P. Wegner. Dimensions of object-based language design. In *Special Issue of SIGPLAN notices*, pages 168–183, October 1987.

[125] Clazien Wezeman. The co-op method, a method for compositional derivation of canonical testers. M.Sc. Thesis, University of Twente, NL, August 1988.

[126] Adam Winstanley. *The elucidation of process-oriented specifications.* PhD thesis, The Queen's University of Belfast, 1992.

[127] N. Wirth. Program development by step-wise refinement. *Comm. ACM*, 14:221–227, 1971.

[128] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

[129] Mario Wolczko. *Semantics of Object-Oriented Languages.* PhD thesis, University of Manchester, 1988.

[130] P. Yelland. First steps towards fully abstract semantics for object oriented languages. In S. Cook, editor, *Proceeedings of the 1989 European Conference on Object Oriented Programming (ECOOP 89)*, pages 347–367. Cambridge University Press (on behalf of British Computer Society), 1989.

[131] Pamela Zave. The operational versus the conventional approach to software development. *Comm. ACM*, 27:104–118, 1984.

Appendix A

Preconditioned Equations: The O-LSTS Model

In this appendix, we define the semantics of preconditions in OO ACT ONE by mapping them to the O-LSTS model. Preconditions are defined for `STRUCTURE` equations, `CLASS` equations and a syntactic sugar defines **total** equations.

Preconditioned Structure Equations

Preconditioned structure equations are defined for transformer, accessor and dual operation as follows.

- **I) Transformer** preconditions, written as:

  $pre_1(P_1, \ldots, P_n) \Rightarrow sc(P_1, \ldots, P_{j-1}).tr(P_j, \ldots, P_n) = newstate_1$ OTHERWISE ...
  $pre_{m-1}(P_1, \ldots, P_n) \Rightarrow newstate_{m-1}$ OTHERWISE
  $newstate_m$, for some $m, n, j \in \{1, 2, \ldots\}, m \geq 2$

  correspond to the parameterised set of **unvalued state-to-state** transitions:

  - $< tr(p_j, \ldots, p_n), newstate_1 > \in From_{sc(p_1, \ldots, p_{j-1})}, \forall p_1, \ldots, p_n$ such that $pre_1(p_1, \ldots, p_n))$
  - $\forall k \in \{2, \ldots, m-2\}$:
    $< tr(p_j, \ldots, p_n), newstate_k > \in From_{sc(p_1, \ldots, p_{j-1})}, \forall p_1, \ldots, p_n$ such that not$(pre_1(p_1, \ldots, p_n))$ and ...and not$(pre_{k-1}(p_1, \ldots, p_n))$ and $(pre_k(p_1, \ldots, p_n))$
  - $< tr(p_j, \ldots, p_n), newstate_m > \in From_{sc(p_1, \ldots, p_{j-1})}, \forall p_1, \ldots, p_n$ such that not$(pre_i(p_1, \ldots, p_n)), \forall i \in \{1, \ldots, m-1\}$

- **II) Accessor** preconditions, written as:

  $pre_1(P_1, \ldots, P_n) \Rightarrow sc(P_1, \ldots, P_{j-1})..acc(P_j, \ldots, P_n) = result_1$ OTHERWISE ...
  $pre_{m-1}(P_1, \ldots, P_n) \Rightarrow result_{m-1}$ OTHERWISE
  $result_m$ for some $m, n, j \in \{1, 2, \ldots\}, m \geq 2$

  correspond to the parameterised set of **valued state-to-state** transitions:

- $< tr(p_j, \ldots, p_n), result_1, sc(p_1, \ldots, p_{j-1}) > \in ValFrom_{sc(p_1,\ldots,p_{j-1})}, \forall p_1, \ldots, p_n$ such that $pre_1(p_1, \ldots, p_n))$
- $\forall k \in \{2, \ldots, m-2\}$:
  $< tr(p_j, \ldots, p_n), result_k, sc(p_1, \ldots, p_{j-1}) > \in ValFrom_{sc(p_1,\ldots,p_{j-1})}, \forall p_1, \ldots, p_n$ such that not$(pre_1(p_1, \ldots, p_n))$ and ...and not$(pre_{k-1}(p_1, \ldots, p_n))$ and $(pre_k(p_1, \ldots, p_n))$
- $< tr(p_j, \ldots, p_n), result_m, sc(p_1, \ldots, p_{j-1}) > \in ValFrom_{sc(p_1,\ldots,p_{j-1})}, \forall p_1, \ldots, p_n$ such that not$(pre_i(p_1, \ldots, p_n)), \forall i \in \{1, \ldots, m-1\}$

- **III) Dual** preconditions, written as:

  $pre_1(P_1, \ldots, P_n) \Rightarrow sc(P_1, \ldots, P_{j-1}).dual(P_j, \ldots, P_n) = newstate_1$ AND $result_1$ OTHERWISE ...
  $pre_{m-1}(P_1, \ldots, P_n) \Rightarrow newstate_{m-1}$ AND $result_{m-1}$ OTHERWISE $newstate_m$ AND $result_m$, for some $m, n, j \in \{1, 2, \ldots\}, m \geq 2$

  correspond to the parameterised set of **valued state-to-state** transitions:

  - $< tr(p_j, \ldots, p_n), result_1, newstate_1 > \in ValFrom_{sc(p_1,\ldots,p_{j-1})}, \forall p_1, \ldots, p_n$ such that $pre_1(p_1, \ldots, p_n))$
  - $\forall k \in \{2, \ldots, m-2\}$:
    $< tr(p_j, \ldots, p_n), result_k, newstate_k > \in ValFrom_{sc(p_1,\ldots,p_{j-1})}, \forall p_1, \ldots, p_n$ such that not$(pre_1(p_1, \ldots, p_n))$ and ...and not$(pre_{k-1}(p_1, \ldots, p_n))$ and $(pre_k(p_1, \ldots, p_n))$
  - $< tr(p_j, \ldots, p_n), result_m, newstate_k > \in ValFrom_{sc(p_1,\ldots,p_{j-1})}, \forall p_1, \ldots, p_n$ such that not$(pre_i(p_1, \ldots, p_n)), \forall i \in \{1, \ldots, m-1\}$

Preconditioned Class Equations

Preconditioned class equations are similar to preconditioned structure equations. The only difference is that a set of structure variable parameters are replaced by one parameter which represents all class values. For example, consider **transformer** preconditions, written generally as:

$pre_1(P_1, \ldots, P_n) \Rightarrow Class1.tr(P_j, \ldots, P_n) = newstate_1$OTHERWISE ...OTHERWISE $newstate_m$

These preconditioned equations correspond to the parameterised set of **unvalued state-to-state** transitions:

- $\forall Class1 \in US(Class), k \in \{1, \ldots, m-2\}$:
  $< tr(p_j, \ldots, p_n), newstate_k > \in From_{Class1}, \forall p_1, \ldots, p_n$ such that not$(pre_1(p_1, \ldots, p_n))$ and ...and not$(pre_{k-1}(p_1, \ldots, p_n))$ and $(pre_k(p_1, \ldots, p_n))$
- $\forall Class1 \in US(Class), < tr(p_j, \ldots, p_n), newstate_m > \in From_{Class1}, \forall p_1, \ldots, p_n$ such that not$(pre_i(p_1, \ldots, p_n)), \forall i \in \{1, \ldots, m-1\}$

Accessor and dual preconditions are defined similarly.

Total Equations

The behaviour of all the elements of a class in response to an external attribute request can be defined to be equivalent using a preconditioned class equation in which the first precondition is true. For example, `[true] => Class1.tr = sle OTHERWISE ...` specifies that $\forall c \in US(Class), < tr, sle > \in From_c$. OO ACT ONE provides a more concise way of specifying this behaviour: `Class1.tr = sle`. This is called a **total equation**.

Appendix B

Static Analysis of OO ACT ONE

B.1 Preprocessing: Removing Syntactic Sugar

The first step in the static analysis of an OO ACT ONE specification, after the syntax has been checked, is the removal of syntactic sugar. The following syntactic mechanisms have to be removed:

- Modules
- Renaming
- Class Invariants

The diagram in figure B.1 shows the way in which this is achieved.

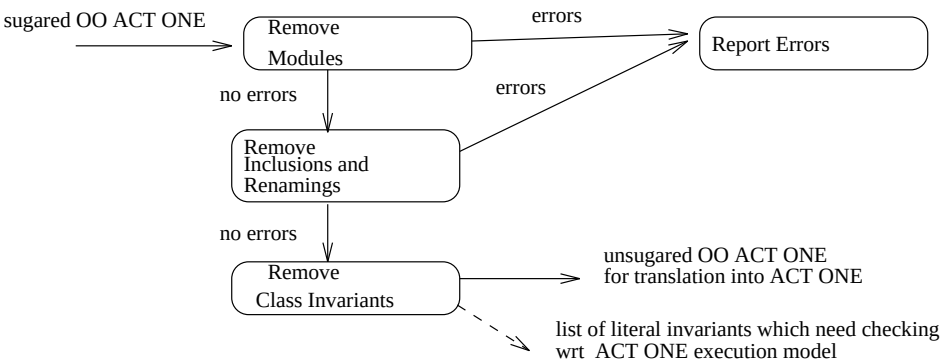


Figure B.1: Preprocessing of OO ACT ONE Syntactic Sugar

Removing Modules

Removing modules is done in three steps:

- I) Check that all modules are uniquely identified. Return an error otherwise.
- II) For every instance of 'MODULE *Module-name*' in a class definition, 'MODULE *Module-name*' is replaced by the list of classes grouped together by the module definition. If *Module-name* is not defined in the specification then an error is reported.

- III) After changing all module references to references to lists of classes, the module definitions are removed.

Removing Inclusions and Renamings

After removing generic class definitions and module definitions, the OO ACT ONE specification is made up of a list of class definitions. These class definitions may be mutually dependent, since one class can be defined to: include the operations and equations of another, or to rename the operations and equations of another. It is not possible, in general, to remove these interdependencies by one single pass through the specification. In a specification with n classes it may take up to n passes through the specification to remove all the interdependencies. Consequently, we define this preprocessing stage as a loop of passes through the specification. The first pass marks all classes which rename or include part of another class. When one class is marked to depend on another class which isn't marked then this class can be redefined (by a simple syntactic relabelling of the appropriate operation labels) and then unmarked. At the end of every pass through the specification the number of classes marked is checked to see if it has decreased. If not, an error is reported. Once all classes are unmarked, all inclusion and renaming mechanisms have been removed.

Removing Class Invariants

Translating class invariants into sets of structure invariants is done in two stages.

- First check that all class invariants are true for all literal values in the classes in which the invariants are defined. This check requires a means of evaluating boolean **state label expressions**. The OO ACT ONE execution model (which formalises the meaning of such an evaluation) is defined by a mapping to ACT ONE. Consequently, this pre-processing stage is defined to generate a list of boolean expressions which must evaluate to true (in the ACT ONE framework of evaluation). When such a list is non-empty, a warning is given to say that class invariants for literal values will be checked at a later stage in the analysis (after the preprocessing is complete). Later, if an executable model has been successfully generated in ACT ONE, the literal requirements (expressed as boolean **state label expressions**) are evaluated and an error is returned if any of the expressions are false.
- Secondly, convert class invariants into sets of structure invariants. For every class definition containing a class invariant, represented as:

  `CLASS` *cname* `USING ...`
  `STRUCTURES:` *st1*,...,*stn* `...`
  `INVARIANTS:` *cname*`1..`*sle*`...EQNS ... ENDCLASS`,

  the class is transformed by the preprocessor into:

  `CLASS ..... INVARIANTS:` *st1* REQUIRES *st1..sle*,..., *stn* REQUIRES *stn..sle*
  `EQNS ... ENDCLASS`

B.2 Static Semantic Checks of Unsugared OO ACT ONE

Static semantic checks of O-LSTS behaviour defined in an OO ACT ONE specification fall into two categories:

- Those which are concerned with 'type checking' equation definitions, and verifying the visibility of classes used in operation definitions. These checks are performed by a static analysis of the ACT ONE produced from the OO ACT ONE specification.
- Other checks are peculiar to the O-LSTS model and cannot be checked across the mapping to ACT ONE.

The remainder of this appendix examines each of these other requirements in turn and gives an overview of the mechanisms which make these checks.

- **Contravariance, Covariance and Subclassing**
  When a subclass is defined to exhibit contravariance and covariance properties with respect to its superclass (or vice versa), additional classification requirements have to be checked:

  - Structure parameters in the subclass must be explicitly defined (in the environment of the new class definition) as subclasses of the corresponding structure parameters in the superclass.
  - Result parameters in the subclass must be explicitly defined (in the environment of the new class definition) as subclasses of the corresponding result parameters in the superclass.
  - Attribute parameters in the subclass must be explicitly defined (in the environment of the new class definition) as superclasses of the corresponding attribute parameters in the superclass.

  To make these checks, it is first necessary to create the explicit class hierarchy for each class. Then, the existence of the required subclassing relationships between subclass and superclass parameters is easily verified. An error is returned if the required relationships are not fulfilled.

  Note that it may not be possible to generate a class hierarchy if the OO ACT ONE is not well defined. For example, one class may be defined in terms of another class which is defined in terms of the original class. This type of circular dependency is checked for when removing the renaming and inclusion syntactic sugar. It is also tested for during the generation of the class hierarchy (in a similar way). An error is returned if the list of classes being analysed do not have a well defined hierarchical structure with respect to the explicit class relationships specified between them.
- **Checking the use of hidden operations**
  As ACT ONE does not facilitate the definition of local operations, it is necessary to check an OO ACT ONE specification to ensure that hidden operations are used only in the class in which they are defined. For every class in an OO ACT ONE specification, the state label expressions

in the equations are analysed to check that operations on classes, other than the one being defined, are not defined as hidden. This analysis is achieved by first producing a list of the hidden operations in each class. An error is returned if a hidden operation of one class is used in the definition of another class.

- **Additional Syntactic Constraints**
  Section 3.2.1.1 defines some additional syntactic constraints for O-LSTS specifications. The constraints specify the way in which string identifiers for state constructors and transition names can be defined. Correspondingly, in OO ACT ONE there are syntactic constraints placed on the naming of operations:

  > All operations must be uniquely categorised (as literal, structure, accessor, transformer or dual) and appear once only in the operation definition.

  Another syntactic constraint placed on the O-LSTS model is that the result of a service request and the newtstate of an object after servicing a service request must be defined using **state label expressions**. Correspondingly, in OO ACT ONE, we require that:

  > The right hand side of equation definitions must be expressed as **state label expressions**.

  This check is carried out as part of the completeness analysis (see below). An error is returned if either of these conditions are not met.
- **The Definition of the Behaviour of a Class is not Distributed Between Other Classes**
  We require that the equations in one class do not specify behaviour for members of another class. Consequently, the left hand side of all equations must be **state label expressions** which have the **server** equal to a member of the class in which the equation is found. This requirement is easily checked by enforcing that all equations have one of the following forms (where `literal` and `structure` are literals or structures respectively of the class in which the equations are defined):

  > `literal..att = ...` or `literal.att = ...` or `structure(...)..att = ...` or `structure(...).a`
  > `= ...`

  This requirement is verified during the completeness analysis (see below).
- **Completeness Analysis**
  The O-LSTS model requires that all states in a class have one, and only one, state transition defined from that state for every attribute of the class. This is called the **completeness condition**. It is more formally defined in section 3.2. Such a requirement cannot be guaranteed through static analysis of the ACT ONE code which is generated from the OO ACT ONE specification.

  The completeness analysis of OO ACT ONE specifications depends on the definition of two new concepts: the **Completeness Set** of a class and the **parameterisation** of an operation. These are defined below.

Definition. Parameterisation:
The parameterisation of an operation, op say, written $Par(op)$:

$Par(op) = op \Leftrightarrow op$ is unparameterised.
Given a parameterised operation, written $op < C_1, \ldots, C_n >$, $Par(op) = op(C_1x_1, \ldots, C_nx_n)$, where
$x_i \in \{1, 2, \ldots\}$ for $i \in \{1, \ldots, n\}$, and $x_i = 1 + x_j$ if $j < i$ and $C_j = C_i$ and $\nexists k \in \{j+1, \ldots, i-1\}$ such that $C_k = C_i$. Otherwise, $x_i = 1$.

Definition. Completeness Set:
The completeness set of a class C, written $CS(C)$, is defined as:

$\{lit.Par(trdl) \mid trdl$ is a transformer or dual of C and lit is a literal operation of $C\} \cup$
$\{lit..Par(accdl) \mid accdl$ is an accessor or dual of C and lit is a literal operation of $C\} \cup$
$\{Par(str).Par(trdl) \mid trdl$ is a transformer or dual of C and str is a structure operation of $C\} \cup$
$\{Par(str)..Par(accdl) \mid accdl$ is an accessor or dual of C and str is a structure operation of $C\}$

We first consider the completeness analysis of classes which are

not defined explicitly as subclasses or superclasses of already existing classes. In a class C which is not defined using the explicit class relationships we require that:

- Given $trans$, a transformer operation of C, either:
  - a) $trans$ is defined by a preconditioned class equation
  - b) $trans$ is partly defined by preconditioned structure equations on a set of structure operations, PS say, and $\forall lit \in$ the set of literal operations of C, $lit.Par(trans) \in CS(C)$ and $\forall st \notin PS$, where st is a structure of C, $Par(str).Par(trans) \in CS(C)$.
- Given acc, an accessor operation of C, either:
  - a) acc is defined by a preconditioned class equation
  - b) acc is partly defined by preconditioned structure equations on a set of structure operations, PS say, and $\forall lit \in$ the set of literal operations of C, $lit..Par(acc) \in CS(C)$ and $\forall st \notin PS$, where st is a structure of C, $Par(str)..Par(acc) \in CS(C)$.
- Given dl, a dual operation of C, either:
  - a) dl is defined by a preconditioned class equation
  - b) dl is partly defined by preconditioned structure equations on a set of structure operations, PS say, and $\forall lit \in$ the set of literal operations of C, $lit.Par(dl) \in CS(C)$ and $lit..Par(dl) \in CS(C)$ and $\forall st \notin PS$, where st is a structure of C, $Par(str).Par(dl) \in CS(C)$ and $Par(str)..Par(dl) \in CS(C)$.
- The expressions on the left hand sides of equation definitions in C do not have any repeated entries. In other words, each equation must be uniquely defined.

Completeness analysis for classes defined using the explicit classification mechanisms is based on the analysis above. In an object based specification, the explicit classification mechanisms define only syntactic sugarings of the inclusion mechanisms. Completeness checks are not concerned with subclassing properties in the OO ACT ONE specification. Consequently, to check the completeness of a class defined using an explicit classification mechanism we generate an intermediate class which exhibits the object based behaviour of the original class but does not include the explicit subclassing mechanism. (The means of generating such a class is similar to the mechanism for removing inclusion syntactic sugar.) This intermediate class is then tested for completeness[1] (as above). It plays no further role after completeness checks terminate.

[1] We accept that more efficient completeness checks can be formulated for classes defined explicitly to exhibit a class relationship with a class which has already been tested for completeness.

Appendix C

Mapping OO ACT ONE to ACT ONE

C.1 Object Based Requirements

I: Classes and Sorts

Every class in an ACT ONE specification is translated into an ACT ONE sort. Each ACT ONE sort is defined inside a type bearing its name. In other words, in the generated ACT ONE code, types are used only as containers for single sorts. Dependencies between classes are mapped into dependencies between the types containing the corresponding sorts. For example,

```
Class USES Class1, ..., Classn
```

is translated into

```
TYPE Class IS Class1, ...Classn SORTS Class OPNS ....
```

The types in the ACT ONE specification are necessary for the modelling of object based dependencies between classes, since in ACT ONE it is not possible to explicitly define dependencies between sorts.

II: Operations

There is a direct correspondence between the operations of an OO ACT ONE class and the operations in the generated ACT ONE code.

- All OO ACT ONE `LITERALS` map to ACT ONE **literal** values. For example, if `lit` is defined as a literal of class `C` then, in the definition of `TYPE C`, there is an operation defined as `lit: -> C`.
- `STRUCTURES` in an OO ACT ONE class `C`, written `st<c1,...,cn>`, map to ACT ONE operations `st: c1,...,cn -> C`.
- `TRANSFORMERS` in class `C` map to ACT ONE operations in two different ways:
  - An unparameterised transformer of `C`, `tr` say, maps to an operation `tr:C -> C`.
  - A parameterised transformer of `C`, `tr<C1,...,Cn>` say, maps to the operation `tr:C, C1,...,Cn -> C`.
- `ACCESSORS` in class `C` also maps to ACT ONE operations in two different ways:

- An unparameterised accessor of `C`, `acc -> C'` say, maps to an operation
  `acc: C -> C (* dual accessor C' *).`
- A parameterised accessor of `C`, `acc<C1,...,Cn> -> C'` say, maps to the operation
  `acc: C, C1,...,Cn -> C (* dual accessor C' *).`

- `DUALS` in class `C` map to ACT ONE operations as follows:
  - An unparameterised dual of `C`, `dl -> C'` say, maps to an operation `dl: C -> C (* dual C' *).`
  - A parameterised accessor of `C`, `dl<C1,...,Cn> -> C'` say, maps to the operation
    `dl: C, C1,...,Cn -> C (* dual C' *).`

III: Hidden Operations

Internal (hidden) operations are mapped as above except that the hidden operations are commented as such in the ACT ONE code. The static analysis of the OO ACT ONE from which the ACT ONE was developed guarantees that hidden operations are used only in the specification of internal behaviour.

IV: Equations

Consider the mapping of total equations, literal equations, unpreconditioned structure equations, preconditioned structure equations and preconditined class equations.

- **1) Total Equations**
  The translation of a total equation from a class definition (`C` say) to a sort definition of the same name (with a **result type** `D` where appropriate), is given below:
  - `C1.tr = sle` → `tr(C1) = sle;`
  - `C1.tr(p1,...,pn) = sle` → `tr(C1,p1,...,pn) = sle;`
  - `C1..acc = sle` → `tr(C1) = dualCD(C1,sle);`
  - `C1..acc(p1,...,pn) = sle` → `tr(C1,p1, ..., pn) = dualCD(C1,sle);`
  - `C1.dl = sle1 AND sle2` → `dl(C1) = dualCD(sle1, sle2);`
  - `C1.dl(p1,...,pn) = sle1 AND sle2` →
    `dl(C1,p1,...,pn) = dualCD(sle1, sle2);`

  Note that the equations generated from the translation are defined in terms of variable parameters. It is a simple, though vital, part of the translation of this and all other equation types to define these variables in the `forall` clause at the beginning of the ACT ONE equation definitions for each sort. We have shown the mappings for all six forms of total equations. The mappings are very similar and so, for conciseness, we consider only a subset of the equation forms in each of the remaining equation type translations.
- **2) Literal Equations**
  The translation of a parameterised dual equation, `dl`, defined on a literal, `lit`, in a class, `C` with **result type** `D` is defined below:
  `lit.dl(p1,...,pn) = sle1 AND sle2` →
  `tr(lit,p1,...,pn) = dualCD(sle1, sle2);`
  The other forms are similarly defined.

- **3) Unpreconditioned Structure Equations**
  The translation of an unpreconditioned unparameterised accessor equation `acc` defined on `str(p1,...,pn)` a structure of class `C` is as follows:
  `str(p1,...,pn)..acc = sle` → `acc(str(p1,...,pn)) = sle;`
  The other forms are similarly defined in an appropriate manner.
- **4) Preconditioned Structure Equations**
  Consider the generic representation of a preconditioned structure equation:

  $pre_1 \Rightarrow sc(P_1, \ldots, P_{j-1}).dl(P_j, \ldots, P_n) = newstate_1$ AND $result_1$
  OTHERWISE ...OTHERWISE
  $pre_{m-1} \Rightarrow newstate_{m-1}$ AND $result_{m-1}$ OTHERWISE
  $newstate_m$ AND $result_m$, for some $m, n, j \in \{1, 2, \ldots\}, m \geq 2$

  This translates into the following set of ACT ONE preconditioned equations:

  $\{MAP(pre_1) \Rightarrow dl(sc(P_1, \ldots, P_{j-1}), P_j, \ldots, P_n) = dualCD(newstate_1, result_1\} \cup$
  $\{MAP(not(pre_1)$ and ...and not(pre_{k-1}) and $(pre_k)) \Rightarrow$
  $dl(sc(P_1, \ldots, P_{j-1}), P_j, \ldots, P_n) = dualCD(newstate_k, result_k) \mid k \in \{2, \ldots, m-2\}\} \cup$
  $\{MAP(not(pre_1)$ and ...and not$(pre_{m-1}) \Rightarrow$
  $dl(sc(P_1, \ldots, P_{j-1}), P_j, \ldots, P_n) = dualCD(newstate_m, result_m)\}$

  The meta-operation MAP defined on the boolean preconditions represents the mapping of the OO ACT ONE **state label expressions** of type Bool to ACT ONE expressions of sort Bool.
- **5) Preconditioned Class Equations**
  Consider the following unparameterised transformer equation defined in class `C` (expressed in generic form):

  $pre_1 \Rightarrow C1.tr = newstate_1$ AND $result_1$ OTHERWISE ...OTHERWISE
  $pre_{m-1} \Rightarrow newstate_{m-1}$ AND $result_{m-1}$ OTHERWISE $newstate_m$ AND $result_m$, for some $m \in \{2, \ldots\}$

  This translates into the following set of ACT ONE preconditioned equations:

  $\{MAP(pre_1) \Rightarrow tr(C1) = newstate_1\} \cup$
  $\{MAP(not(pre_1)$ and ...and not(pre_{k-1}) and $(pre_k)) \Rightarrow tr(C1) = newstate_k \mid k \in$
  $\{2, \ldots, m-2\}\} \cup$
  $\{MAP(not(pre_1)$ and ...and not$(pre_{m-1}) \Rightarrow tr(C1) = newstate_m\}$

V: Structure Invariants Structure invariants generate ACT ONE preconditions which precede every equation defining the behaviour corresponding to the appropriate structure. Consequently, operations on structured objects are defined only when the components of the objects fulfil the precondition property specified by the invariant which generated it.

Additional work is required to map invariant properties in combination with preconditioned equation definitions. Structured preconditions from OO ACT ONE must be coded in ACT ONE as the

boolean conjunction of the corresponding ACT ONE preconditions and the preconditions generated by any invariant properties. Class preconditions pose an even bigger problem than structured preconditions. Static analysis of the ACT ONE code flags every case in which these two mechanisms 'overlap'. The generation of ACT ONE must then include an internal operation which tests an object to see if it is represented as a particular structure expression. All class preconditions are then separated into sets of precondition equations (one for each structure invariant, and one for the remaining objects). Appendix C2, following, illustrates the mapping of preconditions in the `Queue` class.

C.2 Example Queue Behaviour

```
TYPE Queue IS Nat, Boolean
SORTS Queue OPNS
empty:  -> Queue (* Literal *)
Q: Queue, Nat-> Queue (* Structure *)
add:  Queue, Nat -> Queue (* Transformer *)
is-empty:  Queue -> Queue (* Dual accessor Bool HIDDEN *)
rem:  Queue -> Queue (* Dual Nat *)
unspecQueue:  -> Queue
.:  Queue -> Queue
NatResult:  Queue -> Nat
BoolResult:  Queue -> Bool
dualQueueNat:  Queue, Nat -> Queue
dualQueueBool:  Queue, Bool -> Queue
QueueRep:  Queue -> Bool
EQNS FORALL Queue1:  Queue, Nat1, Nat2:  Nat, Bool1:  Bool
OFSORT Queue
add(Queue1, Nat1) = Q(Queue1, Nat1);
add(unspecQueue, Nat1) = unspecQueue;
add(dualQueueNat(Queue1,Nat1), Nat2) = add(Queue1, Nat2);
add(dualQueueBool(Queue1,Nat1), Bool1) = add(Queue1, Nat2);
is-empty(empty) = dualQueueBool(empty, true);
is-empty(Q(Queue1, Nat1)) = dualQueueBool(Q(Queue1, Nat1), false);
rem(empty)= dualQueueNat(empty, unspecNat);
BoolResult(is-empty(Queue1)) =>
rem(Q(Queue1,Nat1))= dualQueueNat(empty, Nat1);
not(BoolResult(is-empty(Queue1))) =>
rem(Q(Queue1,Nat1))= dualQueueNat(Q(.(rem(Queue1)),Nat1),BoolResult(rem(Queue1)));
QueueRep(Queue1) => .(Queue1) = Queue1;
.(dualQueueNat(Queue1, Nat1)) = Queue1;
OFSORT Bool
QueueRep(empty) = true; QueueRep(Q(Queue1, Nat1)) = true;
QueueRep(unspecQueue) = true; QueueRep(dualQueueNat(Queue1,Nat1)) = false
BoolResult(dualQueueBool(Queue1, Bool1)) = Bool1;
OFSORT Nat
NatResult(dualQueueNat(Queue1, Nat1)) = Nat1;
ENDTYPE (* Queue *)
```

C.3 Translating Object Oriented Requirements: An Example

The mapping of object oriented properties to ACT ONE is best illustrated by the following example. The diagram in figure C.1 shows the hierarchy of behaviour which we wish to model in ACT ONE.

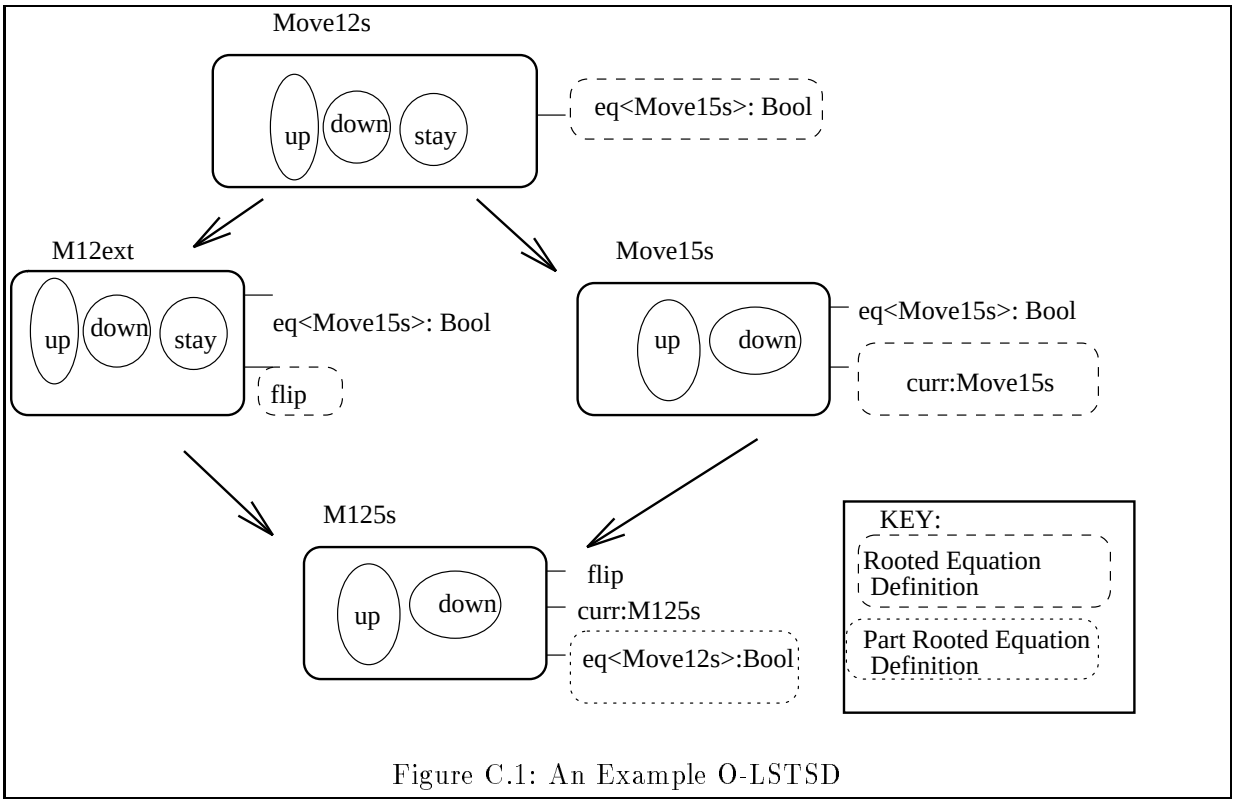


Figure C.1: An Example O-LSTSD

This class hierarchy illustrates two interesting features of object oriented specifications:

- `M125s` has got two direct superclasses (parents). It inherits the `flip` behaviour from `M12ext`, the `curr` behaviour from `Move15s` and the `eq` behaviour partly from `Move12s` (through either of its two parents).
- `M125s` illustrates the rules of contravariance and covariance between subclasses and superclasses. It is defined to return an `M125s` result in response to a `curr` request whilst its superclass `Move15s` is defined to return a superclass of that result class, namely `Move15s`. Furthermore, `M125s` can accept parameter values which are superclasses of the parameter values its superclasses can accept. For example, `M125s` can respond to the request `eq(stay)` but this service is not offered by `Move15s`.

ACT ONE Requirements Model of Class Move12sRoot

```
SPECIFICATION Move12sRoot:noexit
LIBRARY
 Boolean
ENDLIB
TYPE Move12sRoot is Boolean
SORTS Move12s (* using Bool *), Move15s (* superclass Move12s*),
      M12ext (* superclass Move12s *),
      M125s (* superclass M12ext, Move15s *)
OPNS
  (* M125s ------------------------------------------------------ *)
       up,down: -> M125s                    (* literals *)
       flip: M125s -> M125s                 (* transformer from M12ext *)
       eq: M125s, Move12s -> M125s          (* dual Bool part from M12ext *)
           eq: M125s, Move15s -> M125s      (* parameter subclass *)
           eq: M125s, M12ext -> M125s       (* parameter subclass *)
           eq: M125s, M125s -> M125s        (* parameter subclass *)
           M125seq: M125s, Move12s -> M125s (* eq root definition part *)
       curr: M125s -> M125s                 (* dual M125s from Move15s *)
       unspecM125s: -> M125s                (* Unspecified Machinery *)
      (* Dual Machinery *)
       .: M125s -> M125s
       M125sResult: M125s -> M125s
       BoolResult: M125s -> Bool
       dualM125sM125s: M125s, M125s -> M125s
       dualM125sBool: M125s, Bool -> M125s
      (* Subclass machinery *)
       M125stoM12ext: M125s -> M12ext
       M12exttoM125s: M12ext -> M125s
       M125stoMove15s: M125s -> Move15s
       Move15stoM125s: Move15s -> M125s
       M125stoMove12s: M125s -> Move12s
       Move12stoM125s: Move12s -> M125s
      (* Internal Test *)
       M125sRep: M125s -> Bool
  (* Move15s ---------------------------------------------------- *)
      up,down: -> Move15s                       (* literals *)
      eq: Move15s, Move15s -> Move15s           (* dual Bool from Move12s *)
```

```
        eq: Move15s, M125s -> Move15s          (* parameter subclass *)
    curr: Move15s -> Move15s                   (* dual Move15s *)
          Move15scurr: Move15s -> Move15s      (* curr root definition *)
    unspecMove15s: -> Move15s                  (* Unspecified Machinery *)
  (* Dual Machinery *)
    .: Move15s -> Move15s
    BoolResult: Move15s -> Bool
    Move15sResult: Move15s -> Move15s
    dualMove15sBool: Move15s, Bool -> Move15s
    dualMove15sMove15s: Move15s, Move15s-> Move15s
  (* Subclass Machinery *)
    Move15stoMove12s: Move15s -> Move12s
    Move12stoMove15s: Move12s -> Move15s
  (* Internal Test *)
    Move15sRep: Move15s -> Bool
(* M12ext ------------------------------------------------------ *)
    up, down, stay: -> M12ext       (* literals *)
    flip: M12ext -> M12ext          (* transformer *)
    eq: M12ext, Move15s -> M12ext   (* dual accessor Bool from Move12s *)
        eq: M12ext, M125s -> M12ext (* parameter subclass *)
    unspecM12ext: -> M12ext         (* Unspecified Machinery *)
  (* Dual Machinery *)
    .: M12ext -> M12ext
    BoolResult: M12ext -> Bool
    dualM12extBool: M12ext, Bool -> M12ext
  (* Subclass Machinery *)
    M12exttoMove12s: M12ext -> Move12s
    Move12stoM12ext: Move12s -> M12ext
  (* Internal Test *)
    M12extRep: M12ext -> Bool
(* Move12s ----------------------------------------------------- *)
    up, down, stay: -> Move12s                 (* literals *)
    eq: Move12s, Move15s -> Move12s            (* dual accessor Bool *)
        eq: Move12s, M125s -> Move12s          (* parameter subclass *)
        Move12seq: Move12s, Move15s -> Move12s (* eq definition root *)
    unspecMove12s: -> Move12s                  (* Unspecified  Machinery *)
  (* Dual Machinery *)
    .: Move12s -> Move12s
    BoolResult: Move12s -> Bool
    dualMove12sBool: Move12s, Bool -> Move12s
```

```
    (* Internal Test *)
      Move12sRep: Move12s -> Bool

  (* Additional O-LSTS Machinery for Booleans *)
     unspecBool: -> Bool

EQNS FORALL Move15s1, Move15s2: Move15s, Move12s1: Move12s,
            M12ext1: M12ext, M125s1, M125s2, M125s3: M125s, Bool1: Bool
(* M125s ------------------------------------------------------ *)
  (* Inherited from M12ext *)
  OFSORT M125s
  M125sRep(M125s1) =>
   flip(M125s1) = M12exttoM125s(flip(M125stoM12ext(M125s1)));
   flip(dualM125sBool(M125s1, Bool1)) = flip(M125s1);
   flip(dualM125sM125s(M125s1, M125s2)) = flip(M125s1);
  (* Part inherited from M12ext --- contravariance on parameter 1 *)
  M125sRep(M125s1) =>
   eq(M125s1, Move12s1) = M125seq(M125s1, Move12s1);
   M125seq(dualM125sM125s(M125s1,M125s2), Move12s1)= M125seq(M125s1, Move12s1);
   M125seq(dualM125sBool(M125s1,Bool1), Move12s1)= M125seq(M125s1, Move12s1);
   M125seq(up, stay) = dualM125sBool(up,false);
   M125seq(down, stay) = dualM125sBool(down,false);
   M125seq(unspecM125s, stay) = unspecM125s;
  Move15sRep(Move12stoMove15s(Move12s1)) =>
   M125seq(M125s1, Move12s1) =
    eq(M125s1, Move12stoMove15s(Move12s1));
  M125sRep(M125s1) =>
   eq(M125s1, Move15s1) =
    dualM125sBool(
    M12exttoM125s(.(eq(M125stoM12ext(M125s1), Move15s1))),
    BoolResult(eq(M125stoM12ext(M125s1), Move15s1)) );
   eq(M125s1, Move15s1) = eq(M125s1, Move15stoMove12s(Move15s1));
   eq(M125s1, M12ext1) = eq(M125s1, M12exttoMove12s(M12ext1));
   eq(M125s1, M125s2) = eq(M125s1, M125stoMove12s(M125s2));
  (* Inherited from Move15s *)
  OFSORT M125s
  M125sRep(M125s1) =>
   curr(M125s1) =
    dualM125sM125s(
    Move15stoM125s(.(curr(M125stoMove15s(M125s1)))),
```

```
  Move15stoM125s(Move15sResult(curr(M125stoMove15s(M125s1)))) );
 curr(dualM125sBool(M125s1,Bool1))= curr(M125s1);
 curr(dualM125sM125s(M125s1,M125s2))= curr(M125s1);
(* Dual machinery *)
OFSORT M125s
M125sRep(M125s1) =>
 .(M125s1) = M125s1;
 .(dualM125sBool(M125s1, Bool1)) = M125s1;
 .(dualM125sM125s(M125s1, M125s2)) = M125s1;
M125sRep(M125s1) =>
 M125sResult(M125s1) = unspecM125s;
 M125sResult(dualM125sBool(M125s1, Bool1)) = unspecM125s;
 M125sResult(dualM125sM125s(M125s1, M125s2)) = M125s2;
OFSORT Bool
M125sRep(M125s1) =>
 BoolResult(M125s1) = unspecBool;
 BoolResult(dualM125sBool(M125s1, Bool1)) = Bool1;
 BoolResult(dualM125sM125s(M125s1, M125s2)) = unspecBool;
(* Subclass machinery *)
OFSORT M125s
 Move12stoM125s(up) = up; Move12stoM125s(down) = down;
 Move12stoM125s(unspecMove12s) = unspecM125s;
 M12exttoM125s(up) = up; M12exttoM125s(down) = down;
 M12exttoM125s(unspecM12ext) = unspecM125s;
 Move15stoM125s(up) = up; Move15stoM125s(down) = down;
 Move15stoM125s(unspecMove15s) = unspecM125s;
OFSORT Move15s
 M125stoMove15s(up) = up; M125stoMove15s(down) = down;
 M125stoMove15s(unspecM125s) = unspecMove15s;
OFSORT M12ext
 M125stoM12ext(up) = up; M125stoM12ext(down) = down;
 M125stoM12ext(unspecM125s) = unspecM12ext;
OFSORT Move12s
 M125stoMove12s(up) = up; M125stoMove12s(down) = down;
 M125stoMove12s(unspecM125s) = unspecMove12s;
(* Internal Test *)
OFSORT Bool
 M125sRep(up) = true; M125sRep(down) = true;
 M125sRep(unspecM125s) = true;
 M125sRep(dualM125sBool(M125s1, Bool1)) = false;
```

```
   M125sRep(dualM125sM125s(M125s1, M125s2)) = false;
(* Move 15s ------------------------------------------------------ *)
  (* Root definitions *)
  OFSORT Move15s
   curr(Move15s1) = Move15sCurr(Move15s1);
   Move15sCurr(up) = dualMove15sMove15s(up,up);
   Move15sCurr(down) = dualMove15sMove15s(down,down);
   Move15sCurr(unspecMove15s) = unspecMove15s;
   Move15sCurr(dualMove15sBool(Move15s1,Bool1)) = Move15sCurr(Move15s1);
   Move15sCurr(dualMove15sMove15s(Move15s1,Move15s2)) =
    Move15sCurr(Move15s1);
  (* Inherited from Move12s *)
  OFSORT Move15s
   Move15sRep(Move15s1) =>
    eq(Move15s1, Move15s2) =
    dualMove15sBool(
    Move12stoMove15s(.(eq(Move15stoMove12s(Move15s1), Move15s2))),
    BoolResult(eq(Move15stoMove12s(Move15s1), Move15s2)) );
    eq(dualMove15sBool(Move15s1, Bool1), Move15s2) =
     eq(Move15s1, Move15s2);
    eq(dualMove15sMove15s(Move15s1, Move15s2), Move15s2) =
     eq(Move15s1, Move15s2);
    eq(Move15s1, M125s1) = eq(Move15s1, M125stoMove15s(M125s1));
  (* Dual machinery *)
  OFSORT Move15s
   Move15sRep(Move15s1) =>
    .(Move15s1) = Move15s1;
    .(dualMove15sBool(Move15s1, Bool1)) = Move15s1;
    .(dualMove15sMove15s(Move15s1, Move15s2)) = Move15s1;
   Move15sRep(Move15s1) =>
    Move15sResult(Move15s1) = unspecMove15s;
    Move15sResult(dualMove15sMove15s(Move15s1, Move15s2)) = Move15s2;
    Move15sResult(dualMove15sBool(Move15s1, Bool1)) = unspecMove15s;
  OFSORT Bool
   Move15sRep(Move15s1) =>
    BoolResult(Move15s1) = unspecBool;
    BoolResult(dualMove15sMove15s(Move15s1, Move15s2)) = unspecBool;
    BoolResult(dualMove15sBool(Move15s1, Bool1)) = Bool1;
  (* Subclass machinery *)
  OFSORT Move15s
```

```
    Move12stoMove15s(up) = up; Move12stoMove15s(down) = down;
    Move12stoMove15s(unspecMove12s) = unspecMove15s;
  OFSORT Move12s
    Move15stoMove12s(up) = up;  Move15stoMove12s(down) = down;
    Move15stoMove12s(unspecMove15s) = unspecMove12s;
  (* Internal Test *)
  OFSORT Bool
    Move15sRep(up) = true; Move15sRep(down) = true;
    Move15sRep(unspecMove15s) = true;
    Move15sRep(dualMove15sBool(Move15s1, Bool1)) = false;
(* M12ext ------------------------------------------------------ *)
  (* Root definition *)
  OFSORT M12ext
   flip(unspecM12ext) = unspecM12ext;
   flip(dualM12extBool(M12ext1, Bool1)) = flip(M12ext1);
   flip(up) = down; flip(down) = up; flip(stay) = stay;
  (* Inherited from Move12s *)
  OFSORT M12ext
   M12extRep(M12ext1)=>
    eq(M12ext1, Move15s1) =
     dualM12extBool(
     Move12stoM12ext(.(eq(M12exttoMove12s(M12ext1), Move15s1))),
     BoolResult(eq(M12exttoMove12s(M12ext1), Move15s1)) );
    eq(dualM12extBool(M12ext1, Bool1), Move15s1) =
     eq(M12ext1, Move15s1);
    eq(M12ext1, M125s1) = eq(M12ext1, M125stoMove15s(M125s1));
  (* Dual machinery *)
  OFSORT M12ext
   M12extRep(M12ext1)=>
    .(M12ext1) = M12ext1;
    .(dualM12extBool(M12ext1, Bool1)) = M12ext1;
  OFSORT Bool
   M12extRep(M12ext1)=>
    BoolResult(M12ext1) = unspecBool;
    BoolResult(dualM12extBool(M12ext1, Bool1)) = Bool1;
  (* Subclass machinery *)
  OFSORT M12ext
    Move12stoM12ext(up) = up; Move12stoM12ext(down) = down;
    Move12stoM12ext(stay) = stay;
    Move12stoM12ext(unspecMove12s) = unspecM12ext;
```

```
  OFSORT Move12s
    M12exttoMove12s(up) = up;  M12exttoMove12s(down) = down;
    M12exttoMove12s(stay) = stay;
    M12exttoMove12s(unspecM12ext) = unspecMove12s;
  (* Internal Test *)
  OFSORT Bool
    M12extRep(up) = true; M12extRep(down) = true; m12extrep(stay) = true;
    M12extRep(unspecM12ext) = true;
    M12extRep(dualM12extBool(M12ext1, Bool1)) = false;
(* Move12s ------------------------------------------------------ *)
  (* Root definitions *)
  OFSORT Move12s
    eq(Move12s1, Move15s1) = Move12seq(Move12s1, Move15s1);
    eq(Move12s1, M125s1) = eq(Move12s1, M125stoMove15s(M125s1));
    Move12seq(up, up) = dualMove12sBool(up, true);
    Move12seq(up, down) = dualMove12sBool(up, false);
    Move12seq(down, up) = dualMove12sBool(down, false);
    Move12seq(down, down) = dualMove12sBool(down, true);
    Move12seq(stay, up) = dualMove12sBool(stay, false);
    Move12seq(stay, down) = dualMove12sBool(stay, false);
    Move12seq(unspecMove12s, Move15s1) = unspecMove12s;
    Move12seq(Move12s1, unspecMove15s) = unspecMove12s;
    Move12seq(dualMove12sBool(Move12s1, Bool1), Move15s1) =
     Move12seq(Move12s1, Move15s1);
  (* Dual machinery *)
  OFSORT Move12s
   Move12sRep(Move12s1) =>
    .(Move12s1) = Move12s1;
    .(dualMove12sBool(Move12s1, Bool1)) = Move12s1;
  OFSORT Bool
   Move12sRep(Move12s1) =>
    BoolResult(Move12s1) = unspecBool;
    BoolResult(dualMove12sBool(Move12s1, Bool1)) = Bool1;
  (* Internal Test *)
  OFSORT Bool
    Move12sRep(up) = true; Move12sRep(down) = true; Move12sRep(stay) = true;
    Move12sRep(unspecMove12s) = true;
    Move12sRep(dualMove12sBool(Move12s1, Bool1)) = false;
ENDTYPE (* MovesRoot *)
(* EXAMPLE State-label expression evaluations*)
```

```
(*
a = eq( up of Move12s,up of Move15s);
b = eq( a, down of Move15s);
c = eq(down of Move12s, down of M125s);
d = eq(c, up of M125s);
e = flip(stay);
f = eq(e, down of Move15s);
g = eq(up of M12ext, up of M125s);
h = flip(g);
i = eq(up of Move15s, up of Move15s);
j = eq(i, .(i));
k = curr(j);
l = eq(j, up of M125s);
m = flip(up of M125s);
n = eq(m, .(m));
o = eq(flip(m), .(m));
p = curr(n);
p = curr(n);
q= curr(o);
-----------------------------------
a = dualMove12sBool(up, true);
c = dualMove12sBool(down, true);
e = stay;
g = dualM12extBool(up, true);
i = dualMove15sBool(up, true);
m = down;
b = dualMove12sBool(up, false);
d = dualMove12sBool(down, false);
f = dualM12extBool(stay, false);
h = down;
j = dualMove15sBool(up, true);
n = dualM125sBool(down, true);
o = dualM125sBool(up, false);
k = dualMove15sMove15s(up, up);
l = dualMove15sBool(up, true);
p = dualM125sM125s(down, down);
q = dualM125sM125s(up, up)
*)
```

Appendix D

An OO ACT ONE Interpretation of Interaction

D.1 Interaction

Objects which are configured are able to interact (in some as yet unspecified way) for their separate behaviours to be combined in the fulfilment of a service request in their containing object. There are two different types of interaction:

- **Master-Slave Relationships.**
  These are modelled, in OO ACT ONE, when a containing object requests its component objects to fulfil services. It is this interaction which is given a formal definition in the O-LSTS semantics in terms of **state label expression** evaluations.
- **Peer-Peer Relationships.**
  When two components of the same containing object interact such that each can request services of the other they are called peer objects. An OO ACT ONE specification can be implemented to exhibit this type of relationship at the code level, but this is not specified in the requirements model.

A master-slave relationship implies a control flow from master to slave in all interactions. Peer-to-peer interactions imply that control flow can occur in either direction. Control flow is a dynamic property of an object oriented system which is an important aspect of design and implementation but does not have a fundamental role in analysis. The same can be said of data flow. Both these terms are widespread in structured methods but are not central to object oriented analysis.

D.2 Data and Control Flow

Data and control flow, which on the surface seem quite different, are very difficult to distinguish without a formal semantics. One of the main difficulties in applying structured analysis techniques is in distinguishing the two concepts, even though they are modelled in different ways. Data flow diagrams and control flow models are prolific in structured analysis methods but are not explicit in our object oriented model. To understand the reason for this it is necessary to consider some examples.

Data and Control Flow Example: A Two Stack System

Consider the OO ACT ONE `System` specification given below.

```
CLASS System USING Stack, Nat OPNS
STRUCTURES: Sys<Stack, Stack>
DUALS: pop -> Nat
TRANSFORMERS: push<Nat>, move
EQNS
Sys(Stack1, Stack2).pop = Sys(Stack1, Stack2.pop) AND Stack2..pop;
Sys(Stack1, Stack2).push(Nat1) = Sys(Stack1.push(Nat1), Stack2);
Sys(Stack1, Stack2).move = Sys(Stack1.pop, Stack2.push(Stack1..pop))
ENDCLASS (* System *)
```

Specifications with static structure (like `System`) are amenable to three structural interpretations:

- **1) Configuration**
  This specification can be interpreted as saying that the two `Stack` components are configured by the `move` attribute. Chapter 4, section 3, formally defines configuration in terms of dependecy and so it is not necessary to consider it in any more detail as part of this example.
- **2) Data Flow**
  An accessor operation on a component object, `comp` say, written `comp..acc ...`, on the right hand side of an equation definition can be interpreted as modelling data flow from `comp` to the containing object. In other words, the `ACCESSOR` (or `DUAL`) service requests model data flow from client to server (in the form of the result returned by the service). A parameterised attribute on a component object `comp` can be interpreted as modelling data flow into `comp`, i.e. an input parameter. Now, if the data flow into one component matches the data flow out of another this can be interpreted as saying that data flows between the two peer components. More formally, a **state label expression** of the form

  `obj1.att1(..., obj2.att2(...), ...)` or `obj1..att1(..., obj2.att2(...), ...)`

  can be interpreted as data flowing internally from `obj2` to `obj1`.

  In the `System` specification a high level interpretation can lead to the statement that that data flows from `Stack1` to `Stack2` in response to a `move` request. Note that we do not say how the information is transferred.

- **3) Control Flow**

  The simplest interpretation of control flow is from client to server (and back again) when the client requests some service from the server. In the O-LSTS model we do not formulate an interpretation of control flow between peer components. In a `System` class we may implement the first `Stack` to be subordinate to the second `Stack`: 'the `move` request is passed on to the second component which requests a `pop` from the first `Stack`'. Contrastingly, the first `Stack` can be implemented to request a `push` operation of the second component. A third option is to have some additional controlling process (object) which mediates between the two `Stacks`. Such decisions are not the realm of analysis and as such OO ACT ONE does not provide an explicit mechanism for defining such properties. Designers and implementers are free to choose which 'less-abstract' interpretation of control flow they take from an OO ACT ONE specification.

Appendix E

Design Issues

E.1 The ParXStack Process Definition

The Par Specification of the extended `Stack` behaviour (`XStack`) is defined as follows.

```
process ParXStack[push,pop,size](SStack: Stack): noexit:=
hide request, response in
StackIn[push, pop, size, request] (0) |[request]|
StackBody [request, response](SStack) |[response]|
StackOut [pop, size, response](0)
where
process StackIn[push, pop, size, request] (ID: Nat): noexit :=
Reqs[push,pop,size,request](ID) |[request]| ReqController[request](ID) where
process Reqs[push,pop,size,request](IDsStackIn:Nat): noexit:=
(push? Nat1:Nat;
( Reqs[push, pop, size, request] (.(inc(IDsStackIn)))
|||
(request!push!Nat1!IDsStackIn; exit)))
[]
(pop;
(Reqs[push,pop, size, request]( .(inc(IDsStackIn)))
|||
(request!pop!IDsStackIn; exit)))
(size;
(Reqs[push,pop, size, request]( .(inc(IDsStackIn)))
|||
(request!size!IDsStackIn; exit)))
endproc (*Reqs*)
process ReqController[request](ServeID:Nat):noexit:=
(request!push?Nat1:Nat!ServeID; ReqController[request](.(inc(ServeID))))
```

```
[]
(request!pop!ServeID; ReqController[request](.(inc(ServeID))))
[]
(request!size!ServeID; ReqController[request](.(inc(ServeID))))
endproc (* ReqController *)
endproc (*StackIn*)
process StackBody[request, response](SStack: Stack): noexit:=
( request!push? Nat1: Nat?ID:Nat;
(StackBody[request, response](.(push(SStack, Nat1)))
|||
(response!push!ID; exit)))
[]
( request!pop?ID:Nat;
(StackBody[request,response](.(pop(SStack)))
|||
(response!pop!NatResult(pop(SStack))!ID; exit)))
( request!size?ID:Nat;
(StackBody[request,response](.(pop(SStack)))
|||
(response!size!NatResult(size(SStack))!ID; exit)))
endproc (*StackBody*)
process StackOut[pop, size, response](CountStackOut: Nat): noexit:=
(response!pop?NatStackOut:Nat!CountStackOut;
pop!NatStackOut; StackOut[pop, response](.(inc(CountStackOut))))
[]
(response!size?NatStackOut:Nat!CountStackOut;
size!NatStackOut; StackOut[pop, response](.(inc(CountStackOut))))
[]
(response!push!CountStackOut;
StackOut[pop, response](.(inc(CountStackOut))))
endproc (* StackOut *)
endproc (* ParStack *)
```

E.2 Two Mappings from OO ACT ONE to an Initial Full LOTOS Design

Given an OO ACT ONE class, `CName` say, with operation definitions:

> `LITERALS`: $lit_1, \ldots, lit_l$
> Unhidden external `TRANSFORMERS`:
> $tr_1 < \ldots >, \ldots, tr_n < Ptr_{n_1}, \ldots, Ptr_{n_m} >$
> Unhidden internal `TRANSFORMERS`:
> $itr_1 < \ldots >, \ldots, itr_o < Pit_{o_1}, \ldots, Pit_{o_p} >$
> Unhidden `ACCESSORS`:
> $acc_1 < \ldots > - > AResult_1, \ldots, acc_p < Pac_{p_1}, \ldots, Pac_{p_q} > \rightarrow AResult_p$
> Unhidden `DUALS`:
> $dl_1 < \ldots > \rightarrow DResult_1, \ldots, dl_r < Pdl_{r_1}, \ldots, Pdl_{r_s} > \rightarrow DResult_r$

we can define the result of applying *MakeRPC* and *MakePar* to `CName` (in E.2.1 and E.2.2, below).

First, some notation is useful:

- $Req_{CName}?p_1 : P_1?\ldots?p_n : P_n$ represents a parameterised event, where Req is an unhidden attribute of the class `CName`, and $P_1, \ldots, P_n$ are the input parameter types of Req.
- $Req_{CName}!p_1!\ldots!p_n$ represents an event, where Req is an unhidden attribute of the class `CName` and $(p_1, \ldots, p_n$ are values of the appropriate sorts.
- $[]_{Req_{CName}}$ represents a parameterised choice of behaviours over the Req atributes of `CName`.
- $[]_{AD_{CName}}$ represents a parameterised choice over the accessor and dual AD attributes of `CName`.
- $[]_{Tr_{CName}}$ represents a parameterised choice over the transformer Tr attributes of `CName`.
- $Result_{AD_{CName}}$ is the ACT ONE sort corresponding to the result class of the AD accessor or dual attribute of `CName`.

E.2.1 The MakePar Mapping

MakePar(`CName`) =

> **process** `ParCName`$[tr_1, \ldots, tr_n, acc_1, \ldots, acc_p, dl_1, \ldots, dl_r]$`(SCName)`: **noexit**: =
> **hide** `request, response`, $itr_1, \ldots, itr_o$ **in**
> `CNameIn`$[tr_1, \ldots, tr_n, acc_1, \ldots, acc_p, dl_1, \ldots, dl_r,$ `request`$, itr_1, \ldots, itr_o]$`(0)` |[`request`]|
> `CNameBody[request, response] (SCName)` |[`response`]|
> `CNameOut`$[acc_1, \ldots, acc_p, dl_1, \ldots, dl_r,$ `response`$]$`(0)`
> **where** ...

Process `CNameIn` is defined as follows:

process CNameIN[$tr_1, \ldots, tr_n, acc_1, \ldots, acc_p, dl_1, \ldots, dl_r, itr_1, \ldots, itr_o$,request](ID:Nat):
noexit:=
Reqs[$tr_1, \ldots, tr_n, acc_1, \ldots, acc_p, dl_1, \ldots, dl_r, itr_1, \ldots, itr_o$,request](ID)
|[request]|
ReqControl[request](ID)
where
process Reqs[...](ID):**noexit**:=
$[]_{Req_{CName}}$
($Req?p_1 : P_1$,?,...,?$p_n : P_n$; (Reqs[...](.(inc(ID))) ||| request!$Req!p_1!\ldots!P_n$!ID)
endproc (* Reqs *)
process ReqControl[request](ID):**noexit** :=
$[]_{Req_{CName}}$
(request!$Req?p_1 : P_1$,?,...,?$p_n : P_n$!ID; ReqControl[...](.(inc(ID))))
endproc (* ReqControl *)

Process CNameOut is defined as follows:

process CNameOut[$acc_1, \ldots, acc_p, dl_1, \ldots, dl_r$, response] (ID: Nat): **noexit**:=
$[]_{Tr_{CName}}$
(response!Tr_{CName}!ID; CNameOut[...](.(inc(ID))))
$[]_{AD_{CName}}$
(response!AD_{CName}?Result:$Result_{AD_{CName}}$!ID;
AD_{CName}! Result; CNameOut[...](.(inc(ID))))
endproc (* CNameOut *)

Process CNameBody is defined as follows:

process CNameBody [request, response](SCName: CName): **noexit**:=
$[]_{Tr_{CName}}$
(request!$Tr_{CName}?p_1 : P_1, \ldots, p_n : P_n$?ID:Nat;
(CNameBody[...](.(Tr_{CName}(SCName,$p_1, \ldots, p_n$)))
||| (response!Tr_{CName}!ID; **exit**))
$[]_{AD_{CName}}$
(request!$AD_{CName}?p_1 : P_1, \ldots, p_n : P_n$?ID:Nat;
(CNameBody[...](.(AD_{CName}(SCName,$p_1, \ldots, p_n$)))
||| (response!AD_{CName}!$Result_{AD_{CName}}Result($ AD_{CName}(SCName,$p_1, \ldots, p_n$)!ID; **exit**)
endproc (* CNameBody *)

E.2.2 The MakeRPC Mapping

MakeRPC(CName) =

process RPCCName[$tr_1, \ldots, tr_n, acc_1, \ldots, acc_p, dl_1, \ldots, dl_r$](SCName): **noexit**: =
$[]_{Tr_{CName}}$

(*Tr*$_{CName}$?p_1 : P_1, ..., p_n : P_n?ID:Nat;
(RPCCName[...](.(*Tr*$_{CName}$(SCName,p_1, ..., p_n)))
))
[]$_{AD_{CName}}$
(*AD*$_{CName}$?p_1 : P_1, ..., p_n : P_n?ID:Nat;
(RPCCNameBody[...](.(*AD*$_{CName}$(SCName,p_1, ..., p_n)));
AD$_{CName}$!*Result*$_{AD_{CName}}$*Result*(*AD*$_{CName}$(SCName,p_1, ..., p_n)!ID);
(RPCCName[...](.(*Tr*$_{CName}$(SCName,p_1, ..., p_n)))
endproc (* RPCCName *)